Software Preservation After the Internet

Dragan Espenschied
Rhizome
USA/Germany
dragan.espenschied@rhizome.org
0000-0003-1968-6172

Klaus Rechert
University of Applied Sciences Kehl
Germany
rechert@hs-kehl.de
0000-0002-2454-4374

Abstract – Software preservation must consider knowledge management as a key challenge. We suggest a conceptualization of software preservation approaches that are available at different stages of the software lifecycle and can support memory institutions to assess the current state of software items in their collection, the capabilities of their infrastructure, and completeness and applicability of knowledge that is required to successfully steward the collection.

Keywords – software preservation, knowledge management

Conference Topics – Sustainability; From Theory to Practice

I. Software preservation as knowledge management

This article considers software preservation as providing *continuous access* to *reproduced performance*: Different from software in active use that exposes a lot of touch points to larger systems and processes, preserved software is kept available in a historicized and more constrained archival context. This (idealized) setting allows to trace and comprehend the capabilities of all kinds of legacy software objects into the future.

Software objects are regarded as having a workable boundary definition, including blurry objects with parts of their resources or performance located remotely [1]. A boundary definition typically applies to software that is in some sense "unique" from a particular point of view. For instance, from the perspective of a data science research project, computational processes developed for the project need to be reproducible; for a museum of digital art, artworks in the collection that were collected at different points in history need to have their performance available for exhibition and research; a memory institution concerned with digital work environments will want to make available legacy productivity software like word processors. In any of these cases the software object in focus can be composed of multiple artifacts including large amounts of adjacent software and dependencies that are out of the preserving party's reach or control, or might be logistically impossible to turn into local artifacts—that's the "outside world." The software object's boundary thus has to be defined in terms of its performance capabilities at a certain point in time.

We're further defining the performance of software and the reproduction of that performance as a continuum that is in sync with the lifecycle of a software object as it moves from being actively developed, then maintained, and finally encapsulated.

We suggest a conceptualization of software preservation approaches that are available at different stages of the software lifecycle and can support memory institutions to assess the current state of software items in their collection, the capabilities of their infrastructure, and completeness and applicability of knowledge that is required to successfully steward the collection. Ideally this conceptualization can serve as a guide for improving the understanding of the complexity of software preservation: Software in its different manifestations—as source code, installable binary, installed / configured binary, and

iPRES 2023: The 19th International Conference on Digital Preservation, Champaign-Urbana, IL, US.

https://hdl.handle.net/2142/121096

remote process—, its different versions—each potentially exposing different characteristics, e.g. adaptations for different markets, languages, and user groups—create a high-dimensional space that is difficult to oversee and makes it hard to navigate towards formulating desired preservation goals. However, as these goals become more defined, gaps in preservation knowledge and capabilities can be identified and addressed with new research and infrastructure building projects.

II. How software is made and preserved

A software object that performs and a software object that has its performance reproduced are identical on the artifact level: both the item in focus (such as a particular executable) and the software environment it is embedded in are identical in both stages, bit for bit. The assessment differs according to the activities required to produce or reproduce the performance—the care work that supports a software object—and the level of connectedness of the object to the world outside its object boundaries.

In the continuum from performance to reproduction of performance, three stages can typically be observed. The activities defining each stage are also available for preservation and are structurally based on different utopias: ideas about what will be done to the software object in the future in the service of preservation.

A. Active Development

When a piece of software is under active development, it is tightly connected to and dependent on the outside world. Through constant modification, which might expand or otherwise change the software's capabilities, interactions with other software in an ever-changing environment are kept intact. No matter if programmers aim to produce discrete versions or follow a rolling release model, this stage contains a whole additional level of performance: the one required to build the software. Only if the build performance succeeds will the actual desired performance of the software become available.

Preservation at this stage is based on the utopia that programmers will work on and constantly adapt the software to keep its tight integration with the outside world functional. This approach offers the greatest flexibility: over time, a software could be transformed from a desktop into a mobile application, take advantage of new kinds of displays and input devices, be connected to the latest data sources, etc., thereby matching expectations of regular users towards regular contemporary software.

The knowledge required to keep active development going is large and not static: as changes in the outside world happen, some knowledge about outdated components will become obsolete while new knowledge about updated components will need to be integrated into the software development process. This suggests that a history of versions of the software object will be too difficult to keep continuously accessible, unless the capacity for preservation grows with every version created. With the software object being changed continuously it can be expected that knowledge of how to operate and evaluate it will need to be adapted as well.

The potential for knowledge sharing among memory institutions or preservation practitioners is very low at this stage, as all knowledge is object-specific, and the activities rather demand an immersion into software development communities.

Versions of the software object created during active development might be used in the maintenance stage.

B. Maintenance

When a software object is maintained rather than actively developed, development activities are reduced and often focused on adjacent tools and patches. Instead of running the whole build process for an object to make it perform in changing environments, small fixes are applied, operating system settings and driver configuration options are tweaked, and possibly other software tools that improve compatibility with legacy software are used to expand the lifespan of the object. A single version or multiple versions of a software could be created during the active development stage with the plan to later maintain them.

Preservation at this stage is based on the utopia that there will be some clever trick available in the future that allows for a legacy software object to be performed. Patches and tweaks need to be developed or existing tools repurposed to account for a static software object being embedded in a highly dynamic environment.

The capacity required in this stage is significantly smaller than that for active development. Specific knowledge about how to operate the software object is much less dynamic than during active development, because no new versions of the object in question are produced. Knowledge about how to interface existing versions with the changing outside world remains not static, just like in active development new information will have to replace obsolete information. However, there is potential for that knowledge to be generalizable. It is likely that certain classes of objects that can benefit from the same tweaks will be identified. For instance, software that requires a CD-ROM drive can be set up with a virtual CD-ROM driver, Adobe Flash software might be made accessible using the ruffle library, etc.

Over time, the software object will gradually lose its connection to the outside world, as configuration options become unavailable with new versions of operating systems, drivers, and utilities, and required tools will appear too difficult to further develop. At some point the software object will become impossible to perform, perhaps with the last resort being legacy hardware running contemporaneous systems.

Knowledge and tools collected in the maintenance phase might be used in the encapsulation phase.

C. Encapsulation

Encapsulation is an option made possible by emulation [2] and dedicated software preservation frameworks [3]. A fixed version of a software artifact, plus all its dependencies, adjacent tools, and external resources are packaged as immutable disk images, file systems, web archives, etc. and performed by an emulator or a set of emulators orchestrated in a simulated network environment.

All interactions with the outside world happen via managed interfaces of a software preservation framework that controls the emulator or set of emulators. For instance, graphics and sound emitted from an emulator are captured and exposed to the outside world, signals from input devices are translated by the preservation framework into signals that are understood by the emulator.

The utopia of encapsulation is that there will always be emulators in the future and software preservation frameworks will continue to be actively developed.

Object specific knowledge is minimized to only be concerned with how to operate the encapsulated software object. Since this object is not supposed to ever change, and will always perform in the same environment, this knowledge is static. What changes over time are emulators and preservation frameworks, which will need to always accommodate current technical architectures and platforms. Any future issues with the reproduced performance of preserved software objects are to be solved on a framework level. This dynamic knowledge about the preservation framework is highly generalizable, ideally the same for all possible objects, and can be widely shared with a large number of peers with different specializations.

III. Improving preservation capacity

Each stage of a software object is described above in ideal and abstract terms. Especially in a preservation context it is quite unlikely that any of the stages will be observed in their pure form, and mixtures are to be expected, depending on the complexity, size and connectedness of the software in question and its setup.

When framing software preservation as activities enacted on software objects, thus as a knowledge management challenge, it becomes necessary to radically reduce the actively available knowledge required to reproduce the performance of software.

A decision that a software object should be preserved usually happens at a time when it doesn't make sense anymore for the original person or team doing the active development to continue that activity. For instance, software objects produced during an artist residency, or a research grant will need to be taken care of when these projects conclude, and the personnel involved need to move on to other projects.

Institutions as well as communities won't be able to collect, connect, and find ways to apply an ever-growing body of information on software over time. As long as it can be assumed that certain desirable properties of current networked computer systems

should persist into the future—that more or less arbitrary parties can participate in and help develop the overall software environment with at least some degree of autonomy—the world outside of a software object will always keep changing. Hence each new class of software being collected and preserved bears the risk of requiring significant amounts of previously unmanaged knowledge. Yet there is only so much documentation a person can read, or a community can uphold as practice.

Looking at just a single software object in isolation, a care approach modeled after active development makes sense. However, within a collection or archive, it imposes a limit on the number of software objects that can receive preservation treatment and on the time this activity can be sustained. The more knowledge is generalizable instead of object specific, the more institutions and communities can support each other and pool their resources to improve preservation capacity for the field as a whole. Hence, the closer software preservation can move objects towards the encapsulation stage, for single items, for collections, and for software overall, the more likely future generations will be able to explore a rich, diverse, and equitable history of software.

IV. Software preservation realism

It is true that spectacular restoration projects were realized working with legacy source code. [4] [5] They also make for exciting stories as typically important figures from the history of computing and specialized communities, often from the enthusiast space, are involved. Yet exactly these inspiring stories should be interpreted as indicators of the risks of relying on active development for software preservation. While it might be possible to recruit highly skilled developers and knowledgeable hobbyists to work on a groundbreaking software object like an influential game or a landmark operating system—alternatively, pay them well enough to do so—, this is unlikely to happen for an under-appreciated artwork, custom research software, or, plainly boring yet essential software as developed for administrative purposes in government and commerce.

Preservation projects focused on active development are also more likely to succeed for the "classic" model of software creating in which programmers work with local source code and locally available libraries to produce a whole piece of software or component to be packaged and shipped via carrier media or a network connection.

Software Development After the Internet works quite differently. Distributed package managers, interpreted computer languages, and "continuous integration" build processes dominate mainstream development practice and afford developers with previously unknown levels of nimbleness and powerful abstractions. Here the build process has become a performance before the performance, with likely as many variables to consider as for the performance of the software object that is being built. Unless it can be demonstrated that the build process actually works, it is not even possible to assess if all the required dependencies and external resources are available in some form. Since packages and libraries from remote repositories can also change without notice, it becomes increasingly difficult to even deliberately delineate versions of the software object that uses them, and temporarily suspending continuous care for a software object runs the risk of opening a knowledge gap that might turn out to be impossible to close later.

Given these considerations it seems reasonable to move out of the active development stage for preservation purposes and only deal with challenges of an object's "main performance" that is available after the build. This even makes sense when considering that there is no technical difference between reproducing a build performance or reproducing any other software performance, meaning that a build process could be moved to the encapsulation stage just like its final product. The practicality of this approach has to be decided on a case-by-case basis. For instance, if a software object requires rebuilding to produce different desired results in its main performance, the build performance could be made reproducible as well. The usual restrictions of encapsulation would apply, in particular the loss of unmediated interaction with the outside world.

In some cases, it might also not be possible to leave active development behind, in particular when the tools and techniques used to build the software are not well understood or difficult to control due to their novelty or because they're highly proprietary and opaque. Keeping active development going for long enough to gather sufficient knowledge to move to the maintenance or encapsulation stage could be

the only way to develop a long-term perspective for certain types of software objects, such as games built with proprietary toolkits, software requiring highly secured proprietary online accounts, or access to proprietary data.

Entering the maintenance stage is attractive when active development is uneconomical, and Shared knowledge can be utilized. Despite the software industry's push to move all users into subscriptions for most products, there is a wealth of knowledge around on how to keep legacy software running and operational past official support times, by tweaking aspects of new systems to cater for the needs of legacy software objects. Sometimes legacy hardware computer setups are available, or systems are deliberately disconnected from the internet to prevent any unintended automatic updates. Many of these tweaks can be abstracted and applied to several objects of a similar technical composition. Overall, maintenance moves the attention of programmers outside of the object to be preserved to the environments it should perform in.

This approach is for the most part offering the same performance and performance quality as active development—a freshly built object will be as responsive and snappy to interact with as a maintained one that is running on a similar system—, yet at some point the connections to the outside world will become impossible to keep going and the object's performance will degrade.

Returning from maintenance to active development with the plan to update the software object once and to then resume with maintenance can be very expensive and risky. Since active development was suspended while the object was maintained, a large knowledge gap might have appeared that needs to be bridged before development can start. It is also hard to predict for how long the result of such a one-time fix will be able to reproduce the desired performance. An update produced with significant effort might become outdated pretty quickly, calling for another potentially expensive active development phase.

The encapsulation stage in many cases relies on products of active development and maintenance: a software object needs to be built to exist in the first place, and maintenance knowledge can be used to construct a suitable environment to package alongside. Once that is done, knowledge can quite cleanly be separated into static object specific and generalizable infrastructure knowledge. Future risk is reduced to the need for suitable emulators being available and the maintenance of emulators' interfaces with the outside world on the framework level. This means the framework level is ideal for collaboration and most knowledge and development effort can be shouldered in concert by otherwise not affiliated actors. Missing features in an emulator or preservation framework can be identified by practitioners, and stakeholder groups or open fundraising efforts can then commission work to the benefit of any software preservation use case. In an ideal world, emulators and preservation frameworks would be the only places that require active development in order to provide continuous access.

Of course, encapsulation in reality has some drawbacks. Software objects that use bleeding edge or proprietary devices and components or data sources will typically not be possible to capture in full right after creation. For instance, at the time of writing, this is true for projects using virtual reality or augmented reality, dealing with software embedded in a highly competitive market with constantly changing devices, development environments, and real-time online services. Ongoing research on singular objects and classes of objects under active development or maintenance is required to understand which features need to be included into emulators and preservation frameworks.

Additionally, accessing software performance via emulators and preservation frameworks will never be as direct as using software under active development, and to some degree, under maintenance. There will always be some layer users need to pass to access a reproduced performance versus a regular performance, because emulators will have to be spun up and configured by the preservation framework to fit the presented object, and noticeable differences in usage conventions and visual design will contribute to an impression of media discontinuity. This means that any encapsulated software object is necessarily historicized.

V. Overcoming long-term limits on managing software knowledge

Extensive documentation on legacy software products is available in the form of printed and electronic books in libraries. Additionally, the preservation community active on the web provides us with a wide variety of internet artifacts containing tips and hints on configuration, usage and, most importantly, repairing non-functional software products. While this wealth of documentation is necessary and useful, as time goes by, it will become less actionable. The information available was prepared for contemporaneous users and omitted lots of knowledge regarded as implicit in its time. This concerns in many cases basic instructions on how to configure and operate systems that are now deemed obsolete and have fallen out of use. As every change in the software landscape potentially adds another layer of knowledge, demanding preservation professionals to make themselves familiar with everything they need to fully understand any software they are supposed to preserve, is unrealistic and ethically questionable. Similarly, preservation professionals should reflect on their reliance on enthusiast communities and creators for keeping knowledge active and easily retrievable.

Fully configured, encapsulated computing environments (in most cases in the form of a disk image combined with instructions on how to connect and start it up in an emulator) already can serve as a technical embodiment of knowledge, as they can be used without having to look up how to construct one from scratch.

As a next step, recordings of knowledgeable users interacting with encapsulated systems inside a preservation framework can be made so they become deterministically replayable in the future [6]. These recordings can be used to automate simple, recurring tasks, for instance, configuring applications or an operating system; a particularly important use-case for software preservation is automated installations of applications requiring user input during setup. Furthermore, these recordings—together with user annotations—can act as executable documentation, allowing users to follow operational steps and if necessary, take over and adapt a recording to similar tasks, which could then be annotated and stored as a new automated task in a library. Users could choose to have these automations executed in the background, for instance on many encapsulated environments that need to be reconfigured in a similar way or watch the execution to learn the steps.

Even though such recordings act as actionable, executable documentation, a potential library of such recordings will quickly grow into a silo that's difficult to maintain, containing highly context sensitive information for which no concept for indexing apart from manual annotations and basic technical metadata currently exists. Over time, it will become highly desirable to interact with legacy systems on an increasingly abstract level. For instance, to not have to learn how to load a file in dozens of different applications that might run on top of a bunch of operating systems with differing user interface conventions, recordings would need to become much more variable than fully deterministic.

Feeding existing recordings to a learning algorithm has the potential to make this abstraction possible, taking advantage of the similarities in user interface design conventions used within certain time periods. If a sophisticated enough model that matches semantically described desired activity and user actions can be created, it might be trained on legacy tutorial screen capture videos released by software vendors or created by user communities as released on YouTube or similar public video sharing platforms.

1. REFERENCES

[1] D. Espenschied and K. Rechert, "Fencing Apparently Infinite Objects," in *Proceedings of the 15th International Conference on Preservation of Digital Objects*, Boston, U.S., 2018.

[2] D. S. Rosenthal, "Emulation & virtualization as preservation strategies," 2015.

[3] E. Cochrane, K. Rechert, J. Oberhauser, S. Anderson, C. Fox, and E. Gates, "Useable Software Forever," *IPres 2022 Glasg. 12—16 Sept. 2022 Www Ipres2022 Scot*.

[4] G. Mastrapa, "The Geeks Who Saved Prince of Persia's Source Code From Digital Death," *Wired*, Mar. 10, 2023. Accessed: Mar. 10, 2023. [Online]. Available: https://www.wired.com/2012/04/prince-of-persia-source-code/

[5] "The ReCode Project — Matthew Epler," Mar. 10, 2023. https://mepler.com/The-ReCode-Project (accessed Mar. 10, 2023).

[6] J. Oberhauser, R. Gieschke, and K. Rechert, “Automation is Documentation: Functional Documentation of Human-Machine Interaction for Future Software Reuse,” *Int. J. Digit. Curation*, vol. 17, no. 1, Art. no. 1, Sep. 2022, doi: 10.2218/ijdc.v17i1.836.