THE SOFTWARE ASPECT OF PRESERVING DIGITAL ART

Cemal Yılmaz

In order to understand the technical challenges and issues regarding the preservation of software-based digital art, one needs to first understand the root causes of the problem. To this end, we begin with a brief description of the hardware and software stack present in today's computing platforms.

At a very high level, the hardware and software stack is organized in a layered manner. At the bottom of this hierarchy, we have the hardware layer. The hardware layer consists of components, such as CPU (central processing unit), RAM (random access memory), and I/O (input/output) devices, and is responsible for executing low-level machine instructions. On top of the hardware layer, we have the software stack, which is organized into three layers; operating system layer, application support layer, and application layer. The ultimate goal of this layered architecture is to provide an increasingly higher level of abstraction from bottom to the top of the hierarchy. That is, each layer in this architecture is responsible for hiding the details of the lower levels from the upper levels. The operating system layer (e.g., Windows 10, Linux, and Unix), which resides right on top of the hardware layer, makes the hardware transparent to the applications, so that applications do not need to deal with the complicated details of the hardware components in order to operate. Similarly, the application support layer, which consists of components, such as development environments, window managers, and user interfaces, makes the underlying operating system transparent to the applications, so that applications are not affected by the changes in the operating system. On top of the application support layer, we have the application layer where the applications we use on a daily basis, such as Web browsers and social media applications, operate.

Software-based digital art typically resides in the application layer, benefiting from all the abstractions provided by the underlying layers. Although this helps develop better, faster, and more reliable software components in art projects, it also creates dependencies between the artwork and the underlying layers. When this is coupled with the fact that the artwork has no control over the hardware and software stack, it amounts to a proliferation of maintenance issues (if care is not taken). In particular, the underlying layers can change. Their interfaces and semantics can change. The way they interact with each other can change. And, they may even become obsolete. In the presence of such changes (note that this is not a question of if, but when), the artwork needs to be maintained in a timely manner by accommodating the changes to ensure a longer term preservation.

In this essay, we introduce the concept of preservability assurance to refer to all activities and tasks, which focus on providing confidence that digital art will have a long-term preservability. Next, we discuss a number of preservability assurance activities. Note that not all of these activities may be meant to be carried out by non-technical stakeholders. Our point of view, however, is that non-technical stakeholders in art projects, such as artist, should at the very least understand the preservability issues and risks, and be knowledgeable, at a high level, about possible solution approaches, so that they can make better managerial decisions when it comes to balancing the preservability concerns with artistic expression. The suggestions made by this work can be developed by researchers, thought by educational institutions, such as universities, and made practical and promoted by art institutions, such as museums.

Last but not least, for this work we are solely concerned with the preservability assurance of software components in art works. As software is quite different from the other artifacts in art projects, such as hardware, the discussions in this paper may not readily be applicable for them.

Archiving vs. Maintaining

Simply archiving the executables and/or the source code belonging to a piece of software may not be enough for preservation as all the dependencies the software has (i.e., the underlying layers, including the hardware layer) may also need to be archived. Although storing the software stack (compared to storing the hardware stack) is relatively easy as software does not wear out, software needs hardware to run on. And, keeping redundant copies of hardware will go as far as the last copy wears out. Regularly maintaining software components in digital art to accommodate changes in the software and hardware stack, especially in the presence of disruptive technologies, is, therefore, a more effective and reliable strategy for ensuring long-term preservation. The longer the maintenance is delayed, the more challenging and costly it will be, thus the more the risks become reality. This is mainly due to the accumulated technical debt (Tom et al., 2013) – a concept used in software engineering to reflect the implied cost of additional rework caused by ignoring issues or implementing easier, but improper solutions for them.

Note that software is intangible. One cannot touch and feel the shape of a piece of software. It simply runs in the background, orchestrating the hardware. Therefore, software can be maintained without modifying its externally visible behavior, thus without at all affecting the artistic expression of the artwork.

Preservability assurance, but when?

Most (if not all) of the related works in the literature solely concerns preserving digital art after it has been created. This, however, seems to be the exactly the same mistake we, as software engineers, used to make. In particular, we used to think that quality, such as preservability, is something that needs to be addressed after the systems have been developed. After decades of failed software projects, we, however, came to the conclusion that this does not work and that quality is something that needs to be addressed right from the beginning. Preservability concerns regarding digital art is no exception. Therefore, preservability assurance shall be an integral part of any digital art project right from the beginning. Waiting until after artwork has been created to address the preservability concerns, can be too little, too late.

Preservability assurance, but by whom?

All the stakeholders in an art project, including the artists, should contribute to the preservability assurance activities, given that preservability is indeed a concern; not all artists may consent to preservation. One can, however, argue that the creativity of an artist should not be restricted due to some technical issues. By all means, we agree with that. We, however, observe that artists often work with technical stakeholders in art projects, such as software engineers. We, therefore, believe that they, as non-technical stakeholders, should be provided with guidelines, approaches, and tools, so that they can make better managerial decisions when it comes to balancing the preservability concerns with artistic expression. It is, indeed, the artists themselves in the end, who will decide the level of importance that should be attributed to the preservability concerns.

Preservability assurance, but how?

In the software industry, it is not uncommon to have non-technical managers – a role, which is most likely to be played by an artist in a digital art project. As expected, they may have difficulties in evaluating the value and/or consequences of the technologies used in development. To help non-technical managers with their business decisions, we, as the software engineering community, have developed the concept of software governance (Chulani et al., 2008). The ultimate goal of software governance is to quantify the business value of each software module or even each line of code, such that non-technical managers can make better decisions or take educated risks. Consequently, similar approaches can also be developed for preservability governance to help non-technical stakeholders in art projects evaluate the benefits, risks, and the costs of the design decisions made during development.

One frequently exercised practice in software governance (and also in other related activities) is to use software metrics (Fenton, 1991), which aim to quantify different quality attributes of software systems. From the perspective of preservability assurance, one example type of software metrics that can be used is portability metrics (Washizaki et al., 2004), quantifying the ability of running the same software in different environments. These metrics can be used to evaluate various characteristics of portability, including installability, replaceability, and adaptability. Furthermore, as many of these metrics can be extracted from source code as well as from documents, such as requirements and design specification documents, they allow the assessment of the preservability risks even at the very early stages of development. Note that portability is important because one way to preserve digital art can be to port it to a different hardware and software stack (by, for example, replacing obsolete layers).

Software governance approaches are typically developed with the needs of especially the non-technical stakeholders in mind. For technical stakeholders in art projects, we also have a wide spectrum of approaches that they can use for preservability assurance. Next, we briefly discuss some of these approaches. Note that, since the requirements in art projects typically come from the artists, they can enforce the types of the approaches to be employed in the project.

From the perspective of software engineering, preservation generally falls into the category of software maintenance (Bennett et al., 2000). And, software maintenance, for the most part, cannot be carried out in the absence of source code. Therefore, it is of at utmost importance to maintain a repository (such as git (Git, 2020)) of not just the codebase, but also the different artifacts, such as documents and test cases, produced during development. All forms of documentations, including software requirements and design specifications, are of great practical importance as the maintenance team for a piece of art is likely to have a high turnover rate.

Considering the nature of digital art projects, as the requirements are likely to change frequently during development, following agile development processes, such as Scrum (Schwaber, 2002), can be a better fit. At a very high level, the motto for agile processes is “Delivery quickly. Change quickly. Change often.” One dilemma, however, is that agile processes value working software over comprehensive documentation (Fowler et al., 2001). Therefore, agile projects typically have little or no emphasis at all on documentation. In agile projects, the source code itself is considered to be the documentation. This necessitates that the source code needs to be clean, simple, and easy to understand, which, in turn, necessitates frequent refactoring – a technique to restructure the internal structure of a piece of software (often for the purpose of improving the maintainability) without changing its externally visible behavior (Mens et al., 2004). Therefore, all the stakeholders in an agile project, including the artists, shall recognize the value of refactoring and accept all the implied costs (e.g., additional time and effort required for refactoring).

Developing and maintaining test cases is also vital as they need to be run to ensure that recent maintenance activities do not adversely affect the functionality and performance of the software system. All forms of testing, including unit testing, integration testing, system testing, performance testing, and the regression testing (Myers et al., 2004), shall be exercised as they address different quality assurance concerns.

Another approach that can be used to check for regression errors is to have some assertions (Rosenblum, 1995) embedded in the source code. In a nutshell, an assertion is a condition that needs to hold true at runtime. Violating a valid assertion indicates that the system deviates from its expected behavior. Another quite practical property of assertions is that they can be turned on and off at will. For example, they are typically turned off before the system is deployed. Therefore, having assertions, especially the ones regarding the critical functionalities of a digital art project, is a good practice as these assertions can be turned on during preservation activities to make sure that these activities do not have any adverse effects on the artwork.

An advanced form of asserting expectations can be achieved by employing the design by contract approach (Mitchell et al., 2001). In this approach, a software module is shipped with a contract, specifying not only what the user should expect from the module, but also what the module expects from the user. The contracts are expressed in the form of preconditions, postconditions, and invariants, specifying what is to be expected before, after, and during the executions of the modules. The contracts are also executable. That is, a contract can be activated to determine whether it is breached at runtime, which indicates that the system does not work the way it is intended. Digital art can be distributed with executable contracts, which not only help detect regression errors during maintenance activities, but also help determine whether an art installation works as expected. Note that, in the presence of a breach, since the parts of the contract being violated will be known, using executable contracts can also help reduce the space of potential root causes for failures, which, in turn, can greatly improve the turnaround time for bug fixes.

Likely contracts can even be automatically discovered by collecting data at runtime and analyzing the collected data using, for example, artificial intelligence and statistical approaches, to infer behavioral patterns (Ernst et al., 2001). The behavior of digital art can then be checked automatically against these observed patterns to increase the level of confidence after the maintenance activities and/or art installations. Although the representativeness of the discovered patterns is restricted by that of the data used for the analysis (i.e., observed behavior may not precisely specify expected behavior), many empirical studies strongly suggest that event rough approximations can be of great practical help to software engineers (Podgurski, 2003).

Another approach that can significantly improve the preservability of software components in digital art is to design and implement these components in a highly cohesive and loosely coupled manner. In software engineering, cohesion describes how strongly the contents of a module are related to each other, whereas coupling (i.e., dependency) describes how strongly a module is related to other modules (Bass et al., 2003). While increasing cohesion helps get well-defined modules, reducing coupling helps these modules to be standalone, both of which play an integral role for preservability assurance. More specifically, it is typically easier to replace a software module with another module providing the same or similar functionalities, when the module to be replaced is a highly cohesive and loosely coupled module.

To materialize these design ideas, software design patterns (Gamma, 1995) can be used. The rationale behind software design patterns stems from a simple observation that there are some reoccurring design problems in software engineering. The ultimate goal of the design patterns is to determine these reoccurring problems, solve them in an efficient and effective manner, and document the solutions, such that they can readily be adopted in different contexts and projects, rather than solving these problems from scratch every time they are faced. As software engineers, we have developed and documented a large number of software design patterns. Not only the existing design patterns can be leveraged in art projects, but also specific design patterns for preservability assurance can be developed.

In addition to the design patterns, we have also developed a wide range of software design principles (Sommerville, 2011). Some of the important design principles from the perspective of preservability assurance are 1) anticipate obsolescence, i.e., plan in advance for potential changes in the hardware and software stack; 2) design for testing and debugging, i.e., design the system, such that testing and debugging can be automated to the extent possible; and 2) design for portability, i.e., design the system, such that it can run on as many different computing platforms as possible by, for example, using open standards rather than proprietary technologies.

When it comes to portability and managing dependencies, perhaps the most effective technology to be used is the virtualization technology (Campbell et al., 2006). At a very high level, virtualization can be defined as running a virtual instance of a system on another system. By using this approach, digital art can be distributed in the form of a virtual machine (an emulated equivalent of a computer system) or in the form of a virtual container (a lighter weight virtualization technology), where all the dependencies, including the hardware and software stack, are pre-installed. Therefore, deploying a virtual machine (which is typically a straightforward task) automatically deploys everything required by the artwork to operate. Note however that virtualization is not a solution for all the issues we have been discussing so far. After all, a virtual machine is good as long as we have a host platform (e.g., a host machine and a host operating system) supporting the virtualization technology used by the machine. That is, virtualization technologies can change over time and they too can become obsolete.

We have so far focused solely on the internal dependencies they may be possessed by a software system. Digital art can also have external dependencies. For example, an artwork may depend on an information source available on the Web or an artistic expression may depend on certain properties of existing technologies (e.g., “the speed of the internet”). As is the case with internal dependencies, the nature of these external dependencies can change over time. For example, the information source, on which an artwork depends, may become obsolete or the characteristics of the data being published by this source may change or the properties of the technologies may change, e.g., the internet can get faster. These changes not only can create some technical issues, but also may greatly harm the artistic expression. To alleviate these issues, one approach that can be used is mocking (Mostafa et al., 2014). In particular, mock objects can be created for the external dependencies that are likely to change and these mock objects can then be distributed with digital art. In this context, although a mock object replaces an original object, it does not nothing but replay pre-recorded results. Going back to the external information source example, a mock object can automatically be created by capturing the messages being exchanged by the art work and the information source and then these messages can be replayed as needed to reproduce the same artistic expression without requiring the presence of the actual information source – a frequently-used approach known as capture and replay (Zeller, 2009).

The software engineering community has for quite a while been dealing with the same or similar issues, with which the art community struggles to ensure the long-term preservation of software-based digital art. We believe that this presents a win-win situation. On one hand, many of the technologies and processes developed by the software engineering community can readily be employed to preserve digital art. On the other hand, the art community can offer us novel problems and challenges to address. In any case, preservability assurance shall be an integral part of any digital art project right from the beginning and art works shall regularly be maintained.

References

Tom, E., Aurum, A.& Vidgen, R. (2013). An Exploration of technical debt. *Journal of Systems and Software*, 86(6), 1498-1516.

Chulani, S., Williams, C. & Yaeli, A. (2008). Software development governance and its concerns. In *Proceedings of the 1st international workshop on software development governance*.

Fenton, N. E. (1991). *Software metrics: A rigorous approach*. Chapman & Hall.

Washizaki, H., Yamamoto H. & Fukazawa, Y. (2004). A metrics suite for measuring reusability of software components. In *5th International Workshop on Enterprise Networking and Computing in Healthcare Industry (IEEE Cat. No. 03EX717)*.

Bennett, K. H. & Rajlich, V. T. (2000). Software maintenance and evolution: A roadmap. In *Proceedings of the Conference on The Future of Software Engineering*.

Git. (2020). https://git-scm.com.

Schwaber, K. & Beedle, M. (2002). *Agile software development with Scrum*. Prentice Hall Upper Saddle River.

Fowler, M., Highsmith, J. & others (2001). *The Agile manifesto*. Software Development, 9(8), 28-35.

Mens, T. & Tourwe, T. (2004). A Survey of software refactoring. IEEE Transactions on Software Engineering, 30(2), 126-139.

Myers, G. J., Badgett, T., Thomas, T. M. & Sandler, C. (2004). The Art of software testing (2nd ed.). Wiley.

Rosenblum, D. S. (1995). A Practical approach to programming with assertions. *IEEE Transactions on Software Engineering*, 21(1), 19-31.

Mitchell, R., McKim, J. & Meyer, B. (2001). *Design by contract: By example*. Addison-Wesley Publishing Company.

Ernst, M. D., Cockrell, J., Griswold, W. G. & Notkin, D. (2001). Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2), 99-123.

Podgurski, A., Leon, D., Francis, P., Masri, W., Minch, M., Sun, J. & Wang, B. (2003). Automated support for classifying software failure reports. In *Proceedings of the 25th International Conference on Software Engineering*.

Bass, L., Clements, P. & Kazman, R. (2003). *Software architecture in practice*. Addison-Wesley Professional.

Gamma, E. (1995). *Design patterns: Elements of reusable object-oriented software*. Pearson Education India.

Somerville, I. (2011). *Software engineering*. Pearson.

Campbell, S. & Jeronim, M. (2006). An introduction to virtualization. In *Applied Virtualization* (pp. 1-15). Intel.

Mostafa, S. & Wang, X. (2014). An empirical study on the usage of mocking frameworks in software testing. In *Proceedings of the 14th international conference on quality software*.

Zeller, A. (2009). *Why programs fail: A guide to systematic debugging*. Elsevier.