



United Nations
Educational, Scientific and
Cultural Organization



Memory of
the World



UNIVERSITÀ
DI PISA

Inria
inventors for the digital world

The Software Heritage Acquisition Process

*A practical guide
on Github*

V1.1



List of the authors

Short title	SWHAP
Full title	Software Heritage Acquisition Process
Authors:	Laura Bussi, Dept. of Computer Science, University of Pisa < l.bussi1@studenti.unipi.it > Roberto Di Cosmo, Software Heritage, Inria and University of Paris < roberto@dicosmo.org > Mathilde Fichen, CNAM, Software Heritage < mathilde.fichen@inria.fr > Carlo Montangero, Dept. of Computer Science, University of Pisa < carlo@montangero.eu > Guido Scatena, Dept. of Computer Science, University of Pisa < guido.scatena@unipi.it >
Date	November 5, 2025
Contact	Roberto Di Cosmo < roberto@dicosmo.org >

Abstract

The source code of landmark legacy software is particularly important: it sheds insights in the history of the evolution of a technology that has changed the world, and tells a story of the humans that dedicated their lives to it. Rescuing it is urgent, collecting and curating it is a complex task that requires significant human intervention. This document is a practical, simplified, step-by-step guide to the Software Heritage Acquisition Process (SWHAP). Its aim is to provide tools and guidelines to properly archive source code of historical and scientific relevance in the Software Heritage Universal Archive. This updated guide builds upon the original SWHAP guide, which was initially published in 2019 as a result of a fruitful collaboration between the University of Pisa and Software Heritage in this area of research, under the auspices of UNESCO. It has been validated using a selection of software source code produced in the Pisa area and at Inria (French National Institute for Research in Digital Science and Technology) over the past 50 years. The original guide remains the reference for the global approach adopted and the conceptual choices made. This updated guide aims to offer a simplified, practical implementation of SWHAP.

Acknowledgments The authors would like to acknowledge UNESCO for providing the funding that supported the development of this guide.

License This work is distributed under the terms of the Creative Commons license CC-BY 4.0

Contents

1	Introduction	2
2	Why preserve legacy source code?	2
3	What if I am stuck or have a question ?	3
4	Requirements and setup	3
4.1	A source code in machine readable format	4
4.2	A Github account	4
4.3	A Unix Console	4
4.4	Git	4
4.5	Connect to GitHub with SSH	5
5	Preparing your source code for archiving	5
5.1	Final Github repository structure	6
5.2	Prepare your code for archival	7
5.3	Set up your working environment	7
5.4	Upload collected files	8
5.5	Fill in the metadata	10
5.6	Upload machine readable source code	12
5.7	(Re-)Create the development History	13
5.8	Create the final repository	15
6	Trigger the Software Heritage Acquisition	15

1) Introduction

The primary goal of this guide is to help anyone archive **legacy source code** into the **Software Heritage universal archive**. By legacy, we mean any source code which has not been developed on a modern software forge (such as Github or Gitlab). Typically the source code can be stored on a private hard drive, a USB stick or even on paper listings and you might be worried it will get lost if not archived properly. The guide focuses on preserving the software **source code**, which we believe is worth preserving for itself. The process does not tackle the execution of this code, or how to deal with emulation systems.

Note that the process aims at preserving legacy source code and related materials in a **digital** format, to ensure long term availability of the curated materials and the possibility to share and present it to a broad audience.

This document builds up on the SWHAP, the **SoftWare Heritage Acquisition Process** (Abramatic, Di Cosmo, and Zacchioli [2]) to rescue, curate and illustrate landmark legacy software source code. The initial version of this guide was published in 2019 as a joint initiative of Software Heritage and the University of Pisa, in collaboration with UNESCO¹. This guide also aims at simplifying the practical

¹See the Software Heritage webpage dedicated to legacy source code <https://www.softwareheritage.org/swhap/> and get

implementation of the SWHAP as proposed by Pisa University in the SWHAPPE (SWHAP Pisa Enactor)².

2) Why preserve legacy source code?

Software is everywhere, binding our personal and social lives, embodying a vast part of the technological knowledge that powers our industry, supports modern research, mediates access to digital content and fuels innovation. In a word, a rapidly increasing part of our collective knowledge is embodied in, or depends on software artifacts.

Software does not come out of the blue: it is written by humans, in the form of software Source Code, a precious, unique form of knowledge that, besides being readily translated into machine-executable form, should also “be written for humans to read” (Abelson and Julie Sussman [1]), and “provides a view into the mind of the designer” (Shustek [5]).

As stated in the Paris Call on Software Source code as Heritage for sustainable development (Report [4]), from the UNESCO-Inria expert group meeting, it is essential to preserve this precious technical, scientific and cultural heritage over the long term.

Software Heritage is a non-profit, multi-stakeholder initiative, launched by Inria in partnership with UNESCO, that has taken over this challenge. Its stated mission is to collect, preserve, and make readily accessible all the software source code ever written, in the Software Heritage Archive. To this end, Software Heritage designed specific strategies to collect software according to its nature (Abramatic, Di Cosmo, and Zacchiroli [2]).

For software that is easily accessible online, and that can be copied without specific legal authorizations, the approach is based on automation. This way, as of September 2024, Software Heritage has already archived more than 18 billion unique source code files from over 300 million different origins, focusing in priority on popular software development platforms like GitHub and GitLab and rescuing software source code from legacy platforms, such as Google Code and Gitorious that once hosted more than 1.5 million projects.

For source code that is not easily accessible online, a different approach is needed. It is necessary to cope with the variety of physical media where the source code may be stored, the multiple copies and versions that may be available, the potential input of the authors that are still alive, and the existence of ancillary materials like documentation, articles, books, technical reports, email exchanges. Such an approach shall be based on a focused search, involving a significant amount of human intervention, as demonstrated by the pioneering works reconstructing the history of Unix (Spinellis [6]) and the source code of the Apollo Guidance Computer (Burkey [3]).

the initial guide on the UNESCO website <https://unesdoc.unesco.org/ark:/48223/pf0000371017>.

²Subscribe at https://sympa.inria.fr/sympa/subscribe/swhap?previous_action=info

3) What if I am stuck or have a question ?

Because we are still developing and improving the SWHAP process you may stumble upon some difficulties, have some doubts on the best practices to adopt or you may just want to suggest an improvement. To do so, you can join our SWHAP mailing list ³ and share your questions and your comments with the community.

We also developed two video tutorials based on the content of this guide ⁴: 1) An introduction to the Software Heritage Acquisition Process 2) A step by step guide to the Software Heritage Acquisition Process

4) Requirements and setup

To start archiving legacy source code in the Software Heritage Archive, the following elements are required:

- 1) A source code in machine readable format
- 2) A Github account
- 3) A Linux Console
- 4) Git
- 5) Connect to Github with SSH

4.1) A source code in machine readable format

If your source code is already stored in a digital machine-readable format, you can skip this step. However, if your source code is not machine-readable (typically your code is a paper listing), a little prework is required so that your code can be ingested in the Software Heritage Archive.

- 1) Use a scanner to digitalize your code. If your code is too long to be scanned in its entirety, select a section that you find most relevant for archiving.
- 2) Convert your code to a machine-readable format, for example by using an OCR tool such as [OCR.space](#)⁵ and paste your code into a text editor.
- 3) Check for any error, correct if needed, and save your code using the file extension linked to the programming language associated with your code.

4.2) A Github account

Source code ingestion into the Software Heritage archives will first require your source code to be uploaded into a public forge first, such as Github or Gitlab. In this guide we will show you how to do it using Github, and you will therefore need a Github account. If you do not already own one, you can easily create it ⁶(<https://github.com/signup>).

³Subscribe at https://sympa.inria.fr/sympa/subscribe/swhap?previous_action=info

⁴Introduction tutorial https://www.youtube.com/watch?v=ZBTpa09P_Ho&t=4s and step-by-step tutorial <https://www.youtube.com/watch?v=xBQ915N6LyI&t=19s>

⁵<https://ocr.space/>

⁶<https://github.com/signup>

4.3) A Unix Console

To properly deposit your source code into the archive, you will need to use the Git versioning management system. You do not need an extensive understanding of Git mechanisms to do so and we will guide you step by step. However, the command lines we will use are written for a Unix exploitation system. If your computer is running on a Unix-like exploitation system (Unix, Linux, MacOS), you can skip this step. If you are using Windows, you can download a Linux subsystem for Windows.

To do so, you can find detailed instructions online⁷. In practice do the following:

- 1) Open Windows PowerShell
- 2) Enter the following command line: `wsl --install`
- 3) Wait for the installation to complete
- 4) Restart your computer
- 5) Re-open Windows PowerShell and open a new Ubuntu tab (clicking on the small + sign on top)
- 6) You will be asked to enter a new user name and password. And that's it, you can start typing linux command lines in your console.

4.4) Git

Git is the versioning system we will use to curate your source code. If you do not have Git installed yet, you will need to install it. From your Linux console enter the following instruction:

```
sudo apt install git -all
```

If it does not work the first time, you may need to first update the local packages index using the following command line:

```
sudo apt-get update
```

4.5) Connect to GitHub with SSH

The archiving process will require you to interact with Github from your Linux console. To do so, you need to establish a secure SSH connexion between Github and your personal computer. You can find detailed instructions on Github's website⁸. If you do not already have a SSH key, here is what you need to do:

- 1) Create a new SSH using this command line `ssh-keygen -t ed25519 -C "john.smith@gmail.com"` using your own email address. Press enter to accept the default repository or adjust as you wish. Enter a passphrase if you wish or leave empty and press enter.
- 2) Then add your newly created SSH key into your ssh-agent. Check that your ssh-agent is running by entering: `eval "$(ssh-agent -s)"`. Then add your key by entering: `ssh-add ~/.ssh/id_ed25519`
- 3) Navigate to the folder where your SSH key is stored. If you use a Windows Linux Subsystem, it should be in `Linux>Ubuntu>home>myname>.ssh`. Open the public key file `id_ed25519.pub` and copy the key
- 4) Now go to your Github account, click on your logo on the top right corner, go the Settings and SSH and GPG keys. Click on New SSH key, enter a name to your key and paste the public key. Click on Add SHS key.

⁷For example on <https://learn.microsoft.com/en-us/windows/wsl/install>

⁸<https://docs.github.com/en/authentication/connecting-to-github-with-ssh>

You are done with the settings and you are now ready to archive your code into the Software Heritage Universal Archive!

5) Preparing your source code for archiving

In order to archive your legacy code on the Software Heritage Universal Archive, you first need to deposit your code on a public forge such as Github or Gitlab, and most of the work we will do in the following steps aims at doing so in a clean way. In this guide we will leverage the most widely used forge, Github. Note that the process could be easily done on any other forge of your choice.

We will provide a step by step guidance, using a dummy software name *MySoftware* as an example.

Wait, why don't I just manually upload my code on Github then? If you just uploaded your source code files on Github the metadata associated with your code would be wrong. For example, if I, Math, uploaded a code initially written in 1987 by Tim Berners Lee on Github, the commit data will tell that I am the author and that the code was written in 2024. That would be obviously wrong. Using Git command lines to upload our source code on Github will allow us to properly set the metadata.

If your source code has several versions we will also reconstruct the version history, using Git to *stack* each version upon the other and make them easier to navigate and compare one to another for future viewers.

5.1) Final Github repository structure

The structure we want to achieve on Github before launching the archival on the Software Heritage archive is the following. We will create a public repository, named after the software you want to archive (here called *MySoftware*). This repository has two branches (see figure 1):

- 1) The *Main* branch contains all your initial materials (*Raw Materials*), your source code in machine readable format (*Source Code*), the relevant Metadata as well as a ReadMe file helping a future visitor to navigate the repository.
- 2) The *SourceCode* branch contains the reconstructed development history of your source code, i.e. each version of your code stacked one upon the other.

Those two branches allow a future viewer to navigate in your legacy code according to two different angles: either browsing through the historical material and its retranscription (*Main* branch), or viewing the code as if it had been developed with a modern versioning system (*Source Code* branch).

Some vocabulary If you are not familiar with Git:

- A *repository* is similar to a folder, a place where you can store your code, your files, and each file's revision history
- A *branch* is a parallel version of your code that is contained within the repository, but does not affect the primary or main branch.

5.2) Prepare your code for archival

Machine-readable code As mentioned earlier, to start the process your code needs to be in a machine-readable format. If the code is only available in non digital form (e.g. printed listings), you can either transcribe it manually (see figure 2), or use a scanner and an OCR (optical character recognition) tool to

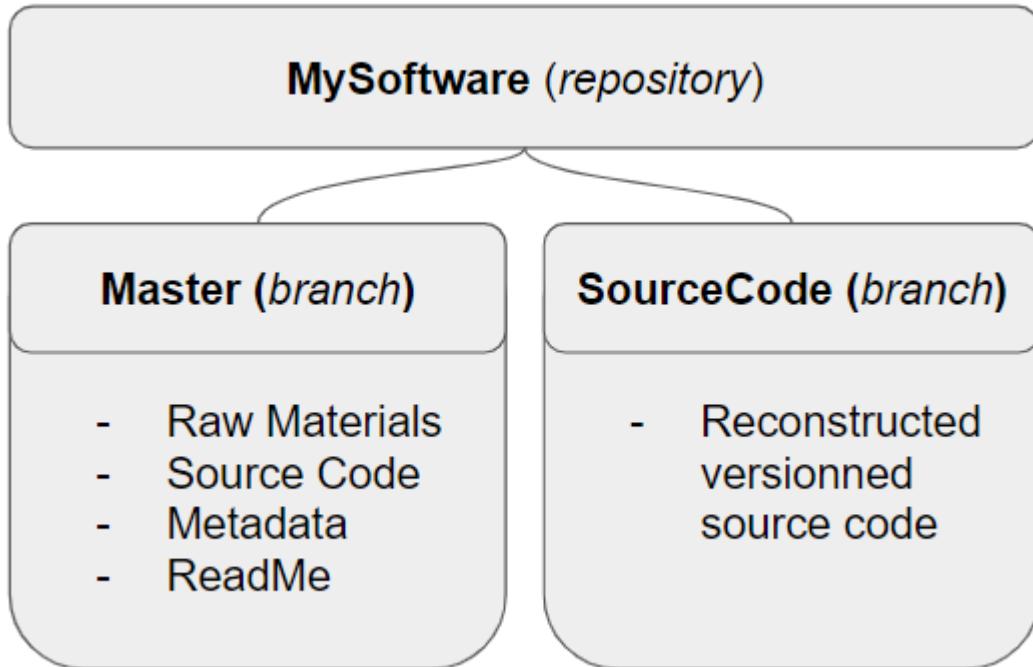


Figure 1: Final Github repository structure.

parse it. In the example below we scanned a paper listing. The scanner had integrated OCR function, so we could copy-past the result in a text editor and correct the errors manually. When saving our edited file, we made sure to correct the file extension to reflect the programming language (in our case .pl).

Unpacking your code & flattening of tarballs If the raw source code is an archived and/or compressed file (.tar or .zip), you should unpack it locally on your computer.

Tarballs distributed by projects often include a single top-level directory (e.g., project-1.0/) to avoid “tarbombs.” For curation purposes, such artificial wrappers should be stripped so that all source files live at the root of the curated version directory.

For historical accuracy purpose we will upload both your source code in its initial format, and in its machine-readable format.

5.3) Set up your working environment

To archive your legacy source code we will be using Github, and we prepared a Github template that you can clone (if you are not familiar with Github lingo *to clone* means *to make a copy*) to create your own working space. Visit the template page on Github⁹, on the upper right hand corner click on Use this template > Create a new repository (see figure 3).

The repository you will create is a temporary working environment, and we recommend naming it MySoftware—Workbench (see figure 4), replace “MySoftware” by the actual name of your software and make it private).

⁹<https://github.com/mathfichen/Swhap-Template>

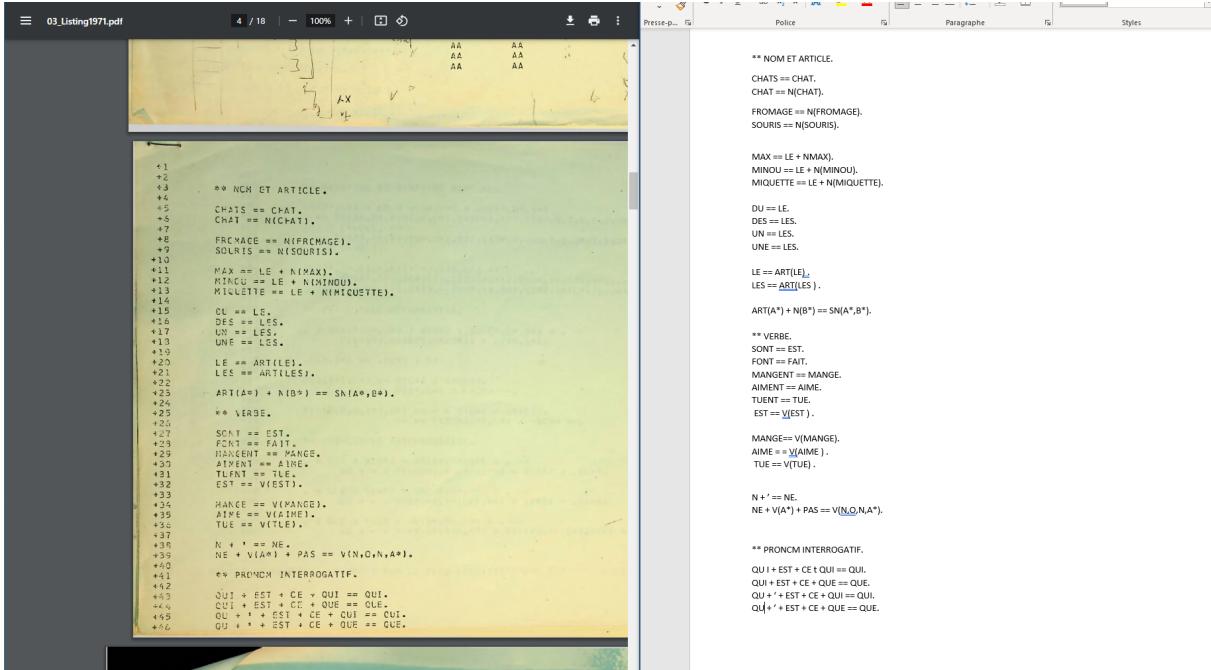


Figure 2: Make your source code machine readable.

Swhap-Template Private template

generated from [Unipa/SWHAP-TEMPLATE](#)

[Sponsor](#) [Unwatch](#) 1 [Fork](#) 0 [Star](#) 0 [Use this template](#)

master 1 Branch 0 Tags [Go to file](#) [Add file](#) [Code](#)

Author	Commit Message	Time
mathfichen	Update README.md	704be55 · 5 minutes ago
.github	Initial commit	12 minutes ago
.swhap	Initial commit	12 minutes ago
metadata	Simplify	9 minutes ago
raw_materials	Initial commit	12 minutes ago
source_code	Simplify	9 minutes ago
README.md	Update README.md	5 minutes ago

About
No description, website, or topics provided.

[Readme](#) [Activity](#) [0 stars](#) [1 watching](#) [0 forks](#)

Releases
No releases published [Create a new release](#)

Figure 3: SWHAP template.

Create a new repository

A repository contains all project files, including the revision history. Already have a project repository elsewhere? [Import a repository.](#)

Required fields are marked with an asterisk (*).

Repository template

 [mathfichen/Swhap-Template](#)

Start your repository with a template repository's contents.

Include all branches
Copy all branches from mathfichen/Swhap-Template and not just the default branch.

Owner *  [mathfichen](#) / MySoftware_Workbench is available.

Great repository names are short and memorable. Need inspiration? How about [sturdy-meme](#) ?

Description (optional)

 **Public**
Anyone on the internet can see this repository. You choose who can commit.

 **Private**
You choose who can see and commit to this repository.

ⓘ You are creating a private repository in your personal account.

Figure 4: Create your Workbench.

Via the Github interface you can edit the Read.me file, using the actual name of your software. To edit a file in Github click on the pencil symbol. When you are done editing, click on Commit to save your changes.

To start working, we create a local copy on our computer, cloning this repository. By clicking on the green button Code, we get a link that we can use for this purpose in the following command from the command line:

```
git clone git@github.com:mathfichen/MySoftware_Workbench.git
```

This command will create local version of the Workbench on your computer, that you can manipulate (add files, edit files, create folders etc) the same way you would usually do it.

In our case (using Linux Subsystem for Windows), the local copy of MySoftware_Workbench has been created at this location:

Linux > Ubuntu > home > mathfichen > MySoftware_Workbench.

Open a Linux command line interpreter and navigate to MySoftware_Workbench. In our case the interpreter current directory is /home/mathfichen, so we just type:

```
cd MySoftware_Workbench
```

5.4) Upload collected files

You are now ready to upload your materials to the Workbench. In your local Workbench, navigate to the raw_materials folder. This folder is meant to store all your initial materials, to help any future viewer understand the origin of the code. This covers the source code in its initial format (scanned listing, compressed file etc.) as well as any contextual element. For example, if the source code was sent over to you by the historical author via email, you can also store this email. You can also store any item you may deem relevant to understand the historical context in which the software was produced, such as technical documentation.

In our case we uploaded two documents: a scanned listing from 1971 and a later digital version from 1972 in a compressed file.

Name	Date modified	Type	Size
Compressed1972.zip	5/6/2024 10:39 AM	Compressed (zipp...)	1 KB
Listing1971.pdf	5/6/2024 10:39 AM	Firefox PDF Docu...	1,411 KB
README.md	5/6/2024 10:39 AM	MD File	1 KB

Figure 5: Add raw materials.

To synchronize our local Workbench with the remote repository, we run the following command lines:

```
git add raw_materials
git commit -m "Added raw materials"
```

```
git push
```

The resulting state of raw_materials in the remote repository is shown in Figure 5.

Name	Last commit message	Last commit date
..		
Compressed1972.zip	aadd zip file	3 hours ago
Listing1971.pdf	aadd zip file	3 hours ago
README.md	Initial commit	last week

Figure 6: Synch raw materials.

5.5) Fill in the metadata

Catalogue

Then navigate to the metadata folder and open the catalogue.md file using any text editor. This file will help any future viewer to better understand the different items you uploaded. Edit the file, filing in the metadata linked to each item your uploaded.

License

Go back to the metadata folder and go to the license .md file and fill in any license information you have about the usage of the software you are archiving.

Version history

Go back once again to the metadata folder and update the version-history.csv folder. The content of this file should correspond to the data you will want to use later on in the process when reconstructing the code synthetic history (see section 5.7 (*Re-)Create the development History*)

CodeMeta

The CodeMeta project defines a standard JSON structure for software metadata. This JSON will allow your code to be more easily discovered by search engines (including the Software Heritage search engine). You can generate such a JSON file using the CodeMeta generator¹⁰. You can see best practices for filling in the data in the paragraph below. Add this JSON file to MySoftware_Workbench>Metadata folder and synchronize with the distant repository.

Synchronize with the remote repository using the following command lines:

```
git add metadata
git commit -m "Updated metadata"
git push
```

¹⁰See the Code Meta project at <https://codemeta.github.io/> and the Code Meta Generator at <https://codemeta.github.io/codemeta-generator/>

Codemeta Best Practices for SWHAP

To ensure consistency and validity against the CodeMeta 2.0 schema, curators should follow these best practices:

- 1. Use only terms defined in the CodeMeta 2.0 context**
 - The context is fixed:

"@context": "<https://doi.org/10.5063/sciencecodemeta/codemeta-2.0>"

 - Avoid custom fields unless explicitly agreed and documented.
- 2. Mandatory fields for SWHAP deposits**
 - @type = "SoftwareSourceCode"
 - name (title of the software)
 - identifier (short unique identifier, e.g. CMM)
 - description (short abstract of the software)
 - version (last archived version)
 - author (primary authors, with affiliation)
 - license (with URL to the license file in metadata/)
 - programmingLanguage (list with details)
- 3. Recommended fields**
 - keywords (research areas, technologies)
 - contributor (secondary contributors)
 - referencePublication (books, papers, manuals)
 - dateCreated, datePublished, dateModified (ISO format YYYY-MM-DD)
 - funder (organization or project supporting the work)
 - codeRepository (URL to original upstream repository or archival location)
 - relatedLink (pointers to SWHAP guide, Software Heritage, project pages)
- 4. Authors and contributors**
 - Use author for primary developers; contributor for collaborators.
 - Include affiliation objects whenever possible (with @type: Organization).
- 5. Licensing**
 - Use a license object with @type: CreativeWork, including name and url.
 - If the license is non-standard (e.g. DEC non-commercial), link directly to the copy in metadata /license.txt.
- 6. Publications**
 - Cite reference documents (manuals, handbooks, articles) under referencePublication, including authors and URLs when available.
- 7. Validation**
 - Always validate codemeta.json using the CodeMeta JSON-LD Playground or JSON schema tools before finalizing.
- 8. Do not include**
 - Fields not defined in CodeMeta 2.0.
 - Internal curatorial notes (keep those in journal.md).

5.6) Upload machine readable source code

We are now going to upload the machine readable versions of your source code into the source_code folder. Each version of the source code should be in a machine readable format, and stored in a dedicated sub-folder.

In our case we create two folders, v1 and v2. v1 contains the transcribed version of our scanned paper

listing from 1971, and v2 contains the unzipped source code from 1972.

When you are done, synchronize with the remote repository:

```
git add source_code
git commit -m "Added machine readable source code"
git push
```

You can check the result in the distant repository.

Name	Last commit message	Last commit date
..		
v1	added machine readable source code	now
v2	added machine readable source code	now
README.md	Initial commit	last week

Figure 7: Add machine readable source code.

Note on Preserving Empty Directories By default, Git does not track empty directories — only files. This creates a problem for SWHAP curation: many historical source archives contain empty directories that are meaningful for build systems, test harnesses, or simply for documenting the project structure. If we ignore them, the curated history in Git and Software Heritage will no longer faithfully represent the original layout.

Workaround.

To preserve empty directories, you should add a placeholder file named `.emptydir` inside each directory that is otherwise empty. This ensures Git records the directory, while making it clear that the file is a curatorial artifact and not part of the original software distribution. These markers should be documented explicitly in the curation metadata, so that future users are aware they were introduced solely for structural preservation.

Example, suppose you find in the original source code tree a couple of empty directories:

```
Tests/ERR/
Tests/REFDIFF/
```

To ensure their correct recording in Git, add inside them an empty file named `.emptydir`:

```
Tests/ERR/.emptydir
Tests/REFDIFF/.emptydir
```

5.7) (Re-)Create the development History

The development history can now be (re-)created either by issuing manually (i.e. for each version directory) the appropriate git commands, or by using a specialized tool. This recreated development history will be stored in a dedicated branch, that we will call `SourceCode`. This branch will be created as an empty orphan branch, meaning that it will be cleaned of any previous content or commits information, as if it were a standalone branch.

Manually

We first create the SourceCode orphan branch

```
git checkout --orphan SourceCode
```

And remove all files and folders:

```
git rm -rf *
```

Then, for every directory of source_code containing a version of the source code, in chronological order, we copy its contents from the master branch to the SourceCode branch, and commit it with the appropriate metadata, as recorded in version_history.csv.

In our case here is how we copy the source contents into our branch:

```
git checkout master -- source_code/v1/*
mv source_code/v1/* .
rm -rf source_code
```

Then we use the following template to create manually an individual commit/release:

```
export GIT_COMMITTER_DATE="YYYY-MM-DD HH:MM:SS"
export GIT_COMMITTER_NAME="Committer Name"
export GIT_COMMITTER_EMAIL="email@address"
export GIT_AUTHOR_DATE="YYYY-MM-DD HH:MM:SS"
export GIT_AUTHOR_NAME="Author Name"
export GIT_AUTHOR_EMAIL=<email@address>
git add -A
git commit -m "Commit Message Here"
```

In our case

```
export GIT_COMMITTER_DATE="2024-05-01 00:00:00"
export GIT_COMMITTER_NAME="Math Fichen"
export GIT_COMMITTER_EMAIL="mathfichen@monadresse.com"
export GIT_AUTHOR_DATE="1972-05-01 00:00:00"
export GIT_AUTHOR_NAME="Colmerauer et al."
export GIT_AUTHOR_EMAIL="<>"
git add -A
git commit -m "V1 of MySoftware"
```

If you do not have an email address for the historical author, you still need to update it with an empty value, to avoid your own default email address to be used instead `export GIT_AUTHOR_EMAIL="<>"`.

We also need to add an annotated tag to this version. For version 1 of MySoftware, here is the command we used, you can adapt it to your needs:

```
git tag -a 1 -m "Version 1"
```

Finally, we clean up the directory before importing a new version

```
git rm -rf *
```

Redo the previous command lines for each version, starting at `git checkout master` — `source_code/v2`. For the last version do not clean up the directory.

Finally, synchronize with your remote repository, creating a new remote `SourceCode` branch.

```
git push --tags origin +SourceCode:SourceCode
```

In your distant repository you will now see a new `SourceCode` branch, that will only display the latest version of your code (See figure 8). The development history of your code can be seen in the *commits* history.

File	Commit Message	Time	Commits
<code>.github</code>	Commit Message Here	5 days ago	
<code>.swhap</code>	Commit Message Here	5 days ago	
<code>mysoftwareV2.pl</code>	Commit Message Here	5 days ago	

With DT2SG If you have numerous source code versions and do not want to reconstruct the development history by hand, the University of Pisa developped a script to do it for you, called DT2SG. This script will automatically used the information stored into `version_history.csv` to perform the successive commits.

Here are the associated Git instructions to run it:

```
dotnet ./DT2SG/DT2SG_app.dll -r mathfichen/MySoftware_Workbench/source_code/ -m  
mathfichen/MySoftware_Workbench/metadata/version_history.csv
```

5.8) Create the final repository

You are now ready to create the final public repository of your Software, that will be ingested into the Software Heritage archive. Go to the Github interface. From the home page, click on the New green button and create a new public repository, named after your software (See Figure 9).

We populate this final MySoftware repository from our workbench.

```
git push --tags git@github.com:mathfichen/MySoftware.git +master:master +SourceCode:  
SourceCode
```

To facilitate the search of the created repository, add the “software-heritage”, “legacy code”, “archive” and “swhap” topic tags to your repository. To do so, click on the setting icon of your repository and add the relevant topics (See figure 10).

Repository template

No template ▾

Start your repository with a template repository's contents.

Owner * **Repository name ***

 mathfichen ▾ / MySoftware

⚠ The repository MySoftware already exists on this account.

Great repository names are short and memorable. Need inspiration? How about [silver-octo-parakeet](#) ?

Description (optional)

 **Public**
Anyone on the internet can see this repository. You choose who can commit.

 **Private**
You choose who can see and commit to this repository.

Initialize this repository with:

Add a README file
This is where you can write a long description for your project. [Learn more about READMEs](#).

Add .gitignore

.gitignore template: **None** ▾

Choose which files not to track from a list of templates. [Learn more about ignoring files](#).

Choose a license

License: **None** ▾

A license tells others what they can and can't do with your code. [Learn more about licenses](#).

ⓘ You are creating a public repository in your personal account.

Create repository

Figure 8: Create final repository.

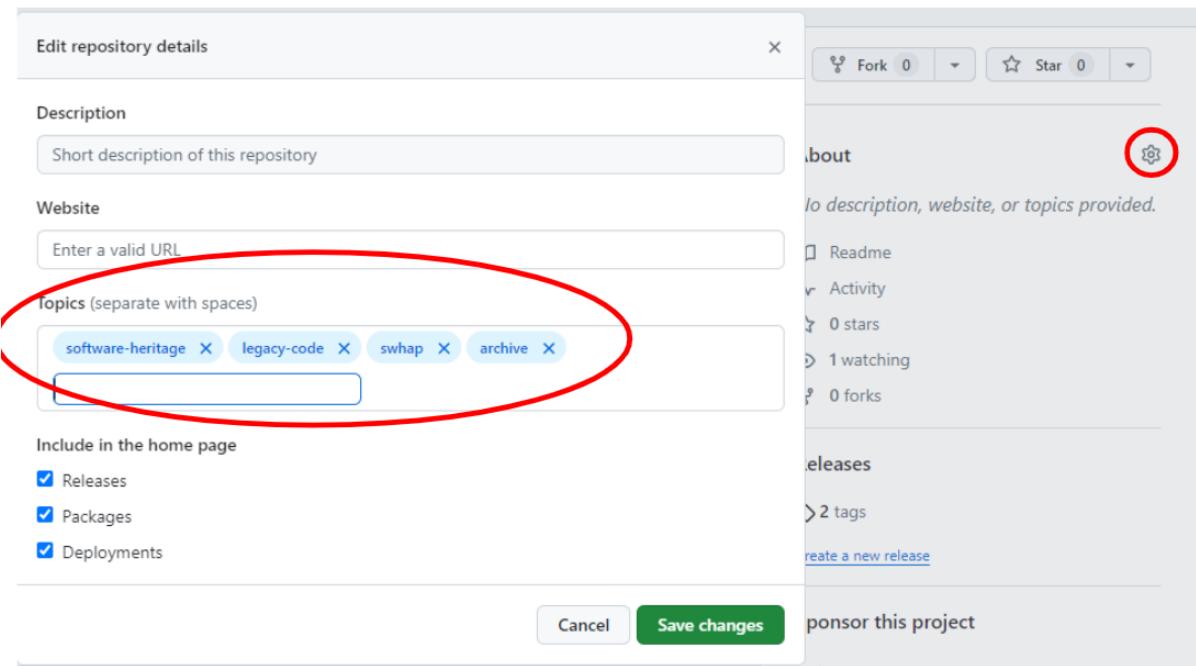


Figure 9: Add topics.

6) Trigger the Software Heritage Acquisition

Even though Software Heritage automatically archives any repository publicly available on Github we suggest you to specifically schedule it to make sure everything runs smoothly. To do so, visit the Software Heritage “Save code now” page¹¹, and submit the URL of your software final repository.

You can contribute to extend the content of the Software Heritage archive by submitting an origin save request. To do so, fill the required info in the form below:

Origin type	Origin url
git	<input type="text" value="https://github.com/mathfichen/MySoftware"/>

Figure 10: View of the *Save Code Now* URL entry bar

You can then follow the archival status of your code in the *Browse Save Request* tab below¹².

Your legacy code is now forever safely archived on the Software Heritage universal archive. You can search for its archive location using its URL in Software Heritage¹³. Your code now has a unique identifier called *SWHID*¹⁴ (Software Heritage IDentifier), that can be used for example to cite your code in an academic

¹¹<https://archive.softwareheritage.org/save/>

¹²<https://archive.softwareheritage.org/save/list/>

¹³<https://archive.softwareheritage.org/browse/search/>

¹⁴See <https://docs.softwareheritage.org-devel/swh-model/persistent-identifiers.html> for more information on the SWHID

paper. This *SWHID* can be found clicking on the Permalink tab on the right side of your archived code page.

Also on the Permalink tab, you can click on the two archived badges and retrieve a markdown code snippet. Use these code snippets in the README of your final software repository. This will display the badges on the first page of your repository, allowing anyone visiting it to click on them and get access to its archive on Software Heritage (See figure 12).

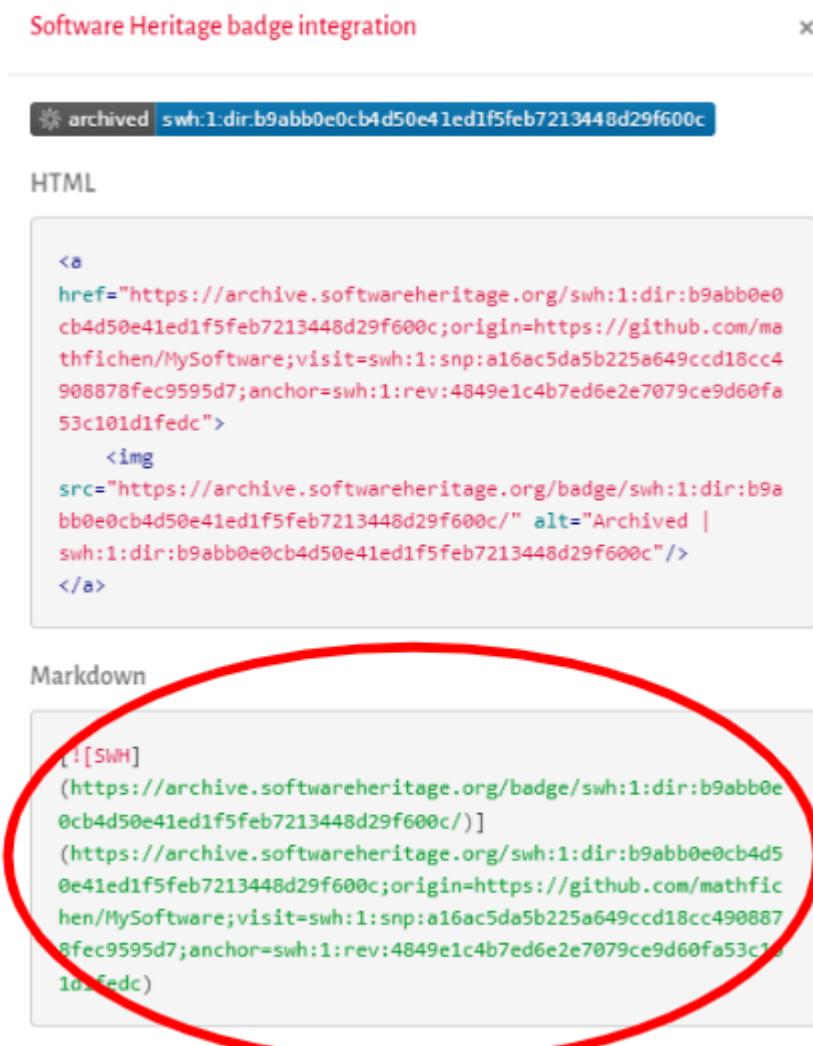


Figure 11: View of the _Permalink_tab

Congrats

Congrats, you are done archiving your code! Please do not hesitate to share your thoughts and send us feedback using the mailing list).

References

- [1] Harold Abelson and Gerald J. Sussman with Julie Sussman. *Structure and Interpretation of Computer Programs*. Cambridge, MA: The MIT Press and McGraw-Hill, 1985, pp. xx + 542. ISBN: 0-262-01077-1 (MIT Press), 0-07-000422-6 (McGraw-Hill).
- [2] Jean-François Abramatic, Roberto Di Cosmo, and Stefano Zacchiroli. “Building the Universal Archive of Source Code”. In: *Commun. ACM* 61.10 (Sept. 2018), pp. 29–31. ISSN: 0001-0782. DOI: 10.1145/3183558. URL: <http://doi.acm.org/10.1145/3183558>.
- [3] Ronald Burkey. *Virtual AGC - Changelog*. Available at <http://ibiblio.org/apollo/changes.html>. Spans years 2003 to 2019.
- [4] Expert Group Report. *Paris Call: Software Source Code as Heritage for Sustainable Development*. Available from <https://unesdoc.unesco.org/ark:/48223/pf0000366715>. 2019.
- [5] Leonard J. Shustek. “What Should We Collect to Preserve the History of Software?” In: *IEEE Annals of the History of Computing* 28.4 (2006), pp. 110–112. DOI: 10.1109/MAHC.2006.78. URL: <http://dx.doi.org/10.1109/MAHC.2006.78>.
- [6] Diomidis Spinellis. “A repository of Unix history and evolution”. In: *Empirical Software Engineering* 22.3 (2017), pp. 1372–1404. DOI: 10.1007/s10664-016-9445-5. URL: <https://doi.org/10.1007/s10664-016-9445-5>.