



United Nations
Educational, Scientific and
Cultural Organization



Memory of
the World



UNIVERSITÀ
DI PISA

Inria
inventors for the digital world

Version 1.1

The Software Heritage Acquisition Process



Software Heritage
THE GREAT LIBRARY OF SOURCE CODE

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> 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	October 6, 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 presents the first version of SWHAP, the Software Heritage Acquisition Process: a protocol for the collection and preservation of software of historical and scientific relevance. SWHAP results from a fruitful collaboration of the University of Pisa with Software Heritage in this area of research, under the auspices of UNESCO, and has been validated on a selection of software source code produced in the Pisa area over the past 50 years.

Acknowledgments L. Bussi wants to acknowledge the Software Heritage Foundation for the scholarship that supported her work and the Department of Computer Science of the University of Pisa for hosting her while working on SWHAPPE.

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

Contents

1	Introduction	2
1.1	General Motivation for using Git and GitHub	3
1.2	SWHAP - GitHub correspondence	3
1.3	Process overview	4
1.4	The SWHAP template	4
1.5	The process, step by step	4
	Instantiation	4
	Collect phase	6
	Notes on OCR	6
	Curate phase	7
1.6	Iteration	8
2	A walkthrough on a running example	9
	Starting the process	9
	Instantiation	9
	Upload files in raw_materials	10
	Unpack the source code in the browsable_source directory	10
	Create Depository	14
	Final depository	14
	Curate the code	14
	(Re-)Create the development History	16
	Create the final repository	18
	Publish the repositories and trigger Software Heritage acquisition	20
	Fill the Workbench metadata	24
	Codemeta Best Practices for SWHAP	24
3	Appendix A - Tools that can help	26
4	Appendix B - A few tips on Github	27

1) Introduction

This document contains the original implementation of the SWHAP process developed at University of Pisa since 2020 using Git and GitHub.

In order to implement SWHAP, the first step is to decide how to instantiate the needed storage and working areas: Warehouse, Depository, Curated source code deposit and Workbench.

For the SWHAP Pisa Enactor (SWHAPPE), the implementation adopted by the SWHAP@Pisa project made the following choices: SWHAPPE exploits the collaborative platform GitHub (<https://github.com/>) as a host platform for all the virtual support areas of the process.

The solutions adopted in SWHAPPE are described in detail in this section, together with their rationale.

1.1) General Motivation for using Git and GitHub

The choice of Git as the designated tool for traceability and historical accuracy, and of GitHub as the unifying platform to support the SWHAP process proceeds from several considerations that we review below.

First of all we discuss the choice of *Git*. One of the key requirements set forth for SWHAP is the need to ensure *full traceability* of the operations performed on the recovered digital assets. This means that each of the virtual places must provide means to record the history of the modifications made to the digital assets, with information on *who did what and when*. It is very convenient to use the same tool in all of the virtual places of the process, as this reduces the learning effort and streamlines the process. All modern version control systems provide the needed functionality, and we have chosen *Git* as our standard tool, as it is open source (another of our requirements) and broadly adopted. *Git* is a powerful tool, and requires some expertise to make the most out of it. However, a large part of the process is scriptable, and this will hide the underlying complexity to the final user, which can then focus on the main issue: curating and preserving the code and its history.

Another important motivation for our choice of Git is the ability to support *historical accuracy*, i.e., providing a faithful view of the history of both the recovered source code and the acquisition process, as prescribed by the SWHAP key requirements. This is properly accommodated by the commit and versioning mechanisms offered by Git, that allow to separate authors from committers: this way on can record both the story of the original software and the story of its curation.

Finally, we had to choose one of the many online platforms that allow to collaborate using *Git*. GitHub, GitLab.com and Bitbucket are the most known ones and are all regularly archived in Software Heritage, so that *long term availability* of their contents is preserved, no matter which one of these platforms is chosen.

Among all these platforms, GitHub is by far the most popular and active, and is also the platform adopted by the University of Pisa, so it was a natural choice, and we believe this will make the learning curve gentler for most SWHAP adopters.

In the following, we provide detailed guidelines to instantiate the process using Git on GitHub. We think that most of what is described in the guide can be easily adapted to any of the other *Git*-based collaborative platforms.

1.2) SWHAP - GitHub correspondence

SWHAPPE is a straightforward implementation of the abstract process, which concretizes the (logical) areas described above by means of *repositories* in GitHub: there are three repositories for each source code acquisition, one for each area of the abstract process:

Workbench repository, to implement the Workbench, i.e. a working area where one can temporarily collect the materials and then proceed to curate the code;

Depository repository, to implement the Depository, where we can collect and keep separated the raw materials from the curated source code;

Source Code repository, to implement the Curated source code deposit, where we store the version history of the code; this version history is usually “synthetic”, rebuilt by the curation team, for old projects that did not use a version control system.

Let’s remark that SWHAPPE has *different* Workbench and Depository repositories for each code acquisition, but it would also be possible to use a single Workbench repository and/or a single Depository repository to work on all the collected software, provided one maintains a well-organised directory structure

which keeps the codes separated. On the other hand, we *need* a Source Code repository for each software project, to be actually ingested in the Software Heritage archive.

1.3) Process overview

GitHub features *template* repositories that can be instantiated whenever needed (see <https://help.github.com/en/articles/creating-a-template-repository>). We used this feature in SWHAPPE, and designed a repository, SWHAP-TEMPLATE, that embodies the core support to enact the process. Its structure and use is shown in figure 2. In the picture and in the following *SWName* is a variable that takes the name of the acquired code as its value at each instantiation.

Once SWHAP-TEMPLATE has been instantiated, the *SWName-Workbench* repository so created need to be cloned to the user's machine, so that he can work on the collected files locally - the Git clone mechanism ensures that these changes can be safely moved to the original repository, for publication and sharing with other actors in the acquisition.

We create two dedicated *branches*¹, that allow to track separately the operations that will be later moved to the Depository and the Development History Deposit: *Depository*, to contain the raw materials and the browsable sources as well as the metadata, and *SourceCode* to organize the source code in view of the reconstruction of its development history. Finally, the *Depository* and *SourceCode* branches become two repositories: the latter is shipped to the Software Heritage archive, the former is published by the organization promoting the acquisition.

Figure 2. Overview of the SWHAPPE process.

1.4) The SWHAP template

The structure of the template is shown in Fig. 3.

First of all, we can see a correspondence between the Depository presented in the process and the area provided by `raw_materials` and `browsable_source`: indeed, these two folders will be moved in order to instantiate the Depository, once they have been loaded, the former with the original materials, just as they have been found or submitted, the latter with a first revision of the source code, made accessible through the GitHub web interface, e.g., archives should be decompressed, code transcribed from pictures, etc.

The `source` folder is provided as the starting point for the creation of the Source Code *Git* repository, in the curation phase. The curator has to recognize each major version of the code, and refactor it accordingly - one separate folder per each version. To create the Source Code Deposit, however, we exploit the *commit* and *versioning* mechanisms of *Git*.

As for the metadata folder, here we record all the information about the software and the acquisition process (catalogue, actors, journal, etc.). The guidelines to fill this part are given in the template itself.

1.5) The process, step by step

Instantiation

The first step is to create an instance of the SWHAP-TEMPLATE², that should be named *SWName-Workbench*, and then to clone it to obtain a local copy on your machine³.

¹More information on Git *branches* can be found in Appendix B.

²See the documentation on <https://help.github.com/en/articles/creating-a-repository-from-a-template>

³See the documentation on <https://help.github.com/en/articles/cloning-a-repository>

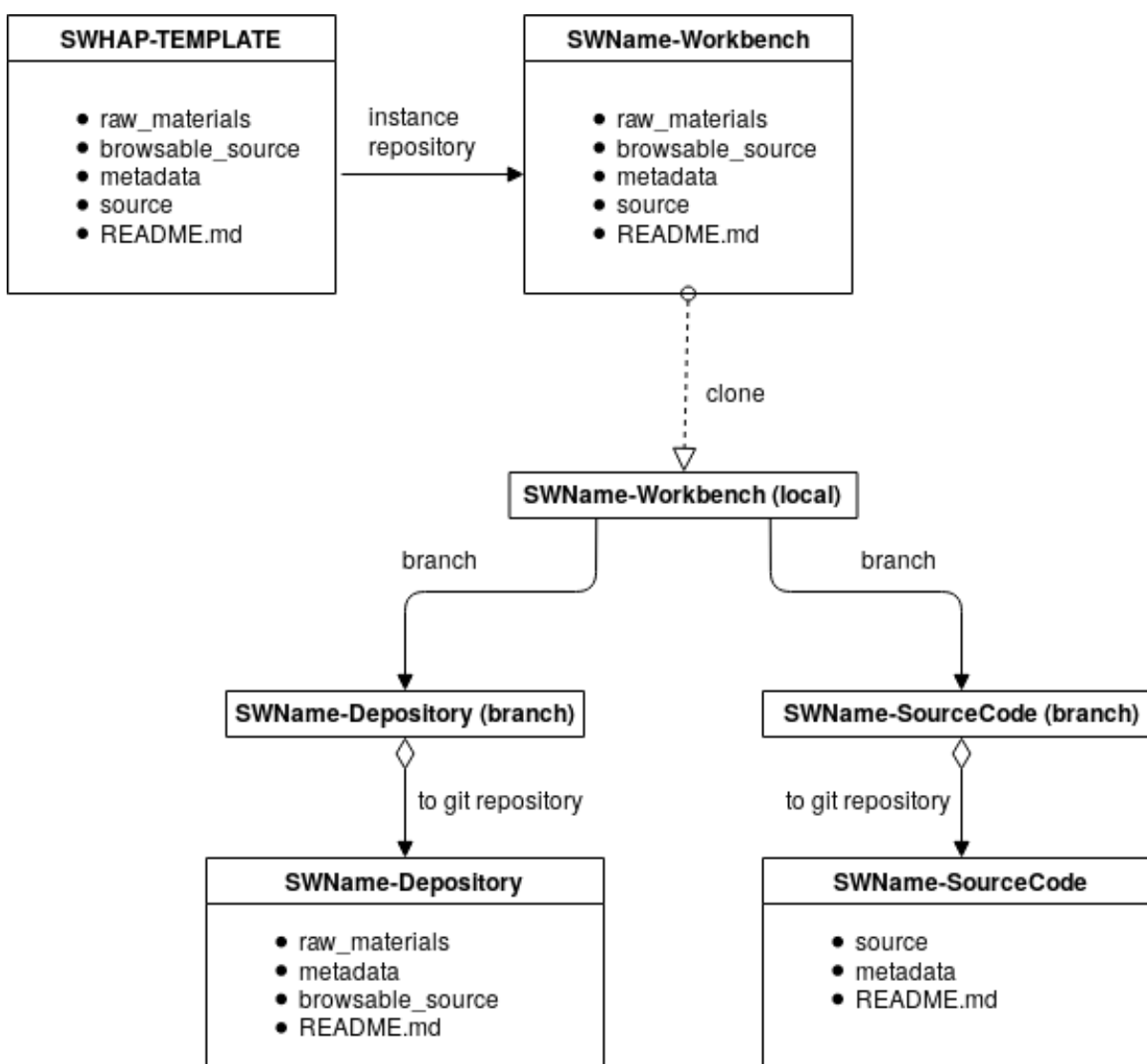


Figure 1: Overview of the SWHAPPE approach

Branch: master ▾	New pull request	Create new file	Upload files	Find File	Use this template	Clone or download ▾
<div>scatenag Update codemeta.json Latest commit 7a8039b 2 hours ago</div> <div> <div>.github</div> <div>Create FUNDING.yml</div> <div>2 days ago</div> </div> <div> <div>browsable_source</div> <div>Minor typos</div> <div>2 days ago</div> </div> <div> <div>metadata</div> <div>Update codemeta.json</div> <div>2 hours ago</div> </div> <div> <div>raw_materials</div> <div>folders README revision</div> <div>20 days ago</div> </div> <div> <div>source</div> <div>typos</div> <div>2 days ago</div> </div> <div> <div>INSTANCE_README.md</div> <div>Update INSTANCE_README.md</div> <div>6 hours ago</div> </div> <div> <div>README.md</div> <div>Added deletion request</div> <div>2 days ago</div> </div>						

Figure 2: Top structure of the Template repository.

From this point on, you'll be able to upload files and to modify/copy/move them locally, and use *Git* commands to push changes to GitHub.

Let us now see the steps to be followed, together with some explanations.

Collect phase

Upload files in `raw_materials`

All the collected files must be uploaded in the `raw_materials` folder.

If there are physical materials, folder `raw_materials` should contain a reference to the related Warehouse, that may follow the Spectrum guidelines [8].

Move the source code to `browsable_source`

All the source code files must then be put into the `browsable_source` folder.

If the raw material is an archive, you need to unpack it locally, put it into the `browsable_source` folder, and upload the result on GitHub by performing a push⁴.

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, SWHAP workbenches 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 (`journal.md`, `README.md`), 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
```

Do not add these markers in the *Depository* branch: only the curated *SourceCode* branch needs them.

Notes on OCR

If the code was only available in non digital form (e.g. printed listings), you can either transcribe it manually, or use a scanner and an OCR (optical character recognition) tool to parse it. See [Appendix A](#) for a list of suggested tools.

⁴See the documentation on <https://help.GitHub.com/en/articles/adding-a-file-to-a-repository-using-the-command-line>.

Particular care should be used to ensure the files in `browsable_source` have the correct extension: scanner and OCR usually generate files with a generic `.txt` extension, that must be changed to the extension typically used for the programming language they contain.

Note that, at this stage, we are not interested in precise information about the versions of the software. The purpose is to have machine-readable documents.

Finally, in preparation for the curation phase, you may want to copy the files in `browsable_source` to the `source` folder.

Create Depository

The next step is to create the branch Depository, containing only the folders `raw_materials` and `browsable_source`, together with the metadata updated to this point. Then, create the Depository repository from this branch.

Curate phase

Curate the source code

Once the Depository creation is complete, you can move back to the `source` folder in the `main` branch. Here you have to divide and number the versions, putting the files of each one in a dedicated folder.

Flattening of Tarballs 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.

Determining who did what and when In practice, this means that *for each version of the software* you need to ascertain:

- the *main contributing author*,
- the *exact date* of the release of this particular version

This information should be consigned in a dedicated metadata file, `version_history.csv`, with the following fields:

Field name	description
directory name	name of the directory containing the source code of this version
author name	name of the main author
author email	email of the main author, when available
date original	original date when this version was made
curator name	name of the curator person or team
curator email	the reference email of the acquisition process
release tag	a tag name if the directory contains a release, empty otherwise
commit message	text containing a brief note from the curation team

(Re-)Create the Development History

Now we are ready to (re-)create the development history of the software. First you need to create a branch Source Code, with the `src` folder.

Then, you can proceed in two ways:

- *manually*: using the *Git* commands to push the successive versions into the [source](#) folder, reading the information collected in the file `version_history.csv` to set the fields for each version to the values determined during the curation phase;
- *automatically*: using a tool that reads the information from `version_history.csv` and produces the synthetic history in a single run; one such tool has been developed, DT2SG (<https://github.com/Unipisa/DT2SG>) , and you can see a running example in the next section.

The result will be a branch that materializes the development history of the software via Git commits and releases.

Create the final repository

Finally you can create the “official” software repository, taking the versions history from the `src` branch and the metadata from the `main` branch.

1.6) Iteration

New material may be discovered after the process has been completed, triggering an iteration of some of the phases described above. In this case, we recommend to proceed as follows:

- if new raw material (non-source code) is found, we have to clone the Depository repository and add new items to it. In this way, the performed commits will correctly follow the previous ones.
- if new source code is found, after we collected it in the Depository, we have the following cases:
 - (1) The recovered source code is related to a version which is already included in the software history.
 - (2) The source code represents a completely new version, with respect to the software history as it was previously collected.

We are not finished yet, since in both cases the SourceCode repository is no longer consistent with the collected source code, and we have to recreate it, performing the following steps:

- Delete the SourceCode repository.
- Move back to the Workbench and according to the current case:
 - if (1), add the source code to the correct version.
 - if (2), add the new version folder with the related metadata.
- Recreate the software history as for the first iteration.

2) A walkthrough on a running example

In this section we will show the process at work on one of the first source code acquired by the SWHAP@Pisa project, the CMM conservative garbage collector for C++ that was initially developed for project PoSSo (Polynomial System Solver) and later became the basis for the Java GC and the Oak GC [7]. Since it has evolved through various versions, CMM is a good workbench for SWHAPPE and an appropriate example to show how to use the tools.

Starting the process

The acquisition process of the CMM software started informally when one of the authors, still active in the Computer Science department, learned about the SWHAP project, and proposed to search for the source code and make it available to the project. Shortly after, we received a mail message with all the sources, as well as the associated research article. Since the materials were already in digital form, the process does not involve a Warehouse.


Instantiation

Create a new repository from SWHAP-TEMPLATE

The new repository will start with the same files and folders as [Unipisa/SWHAP-TEMPLATE](#).

Owner

Repository name *


 Unipisa ▾

 /


CMM-Workbench ✓

Great repository names are short and memorable. Need inspiration? How about **expert-computing-machine**?

Description (optional)

☐  **Public**

Anyone can see this repository. You choose who can commit.

☒  **Private**

You choose who can see and commit to this repository.

Create repository from template

Figure 3: Instantiation of the template

We instantiate on GitHub the SWHAP repository template⁵ into a new repository⁶, that we name “CMM-Workbench”. This action, as most of the following ones, can be performed through the user interface (as shown in Figure {fig:temp_instempty citation}), or programmatically through the GitHub API.

It has the same directory structure as SWHAP-Template, as shown in Figure {fig:templateempty citation}.

To facilitate the search of the created repository, we add the “software-heritage”, “workbench” and “swhappe” tags, as shown in Figure {fig:workbench_tagempty citation}.

To start working, we create a local copy on our computer, cloning this repository⁷. By clicking on the green button “clone or download” (Figure {fig:cmm_wb_instempty citation}), we get a link that we can use for this purpose in the following command from the command line:

```
git clone https://github.com/Unipisa/CMM-Workbench.git
```

Now, we have a local copy of the CMM-Workbench, and we can, first of all, update the README.md file with the correct name and description of the acquisition, and synchronize it with the remote repository:

```
git add README.md
git commit README.md -m "Updated README"
git push
```

We are now ready to start the collect phase.

Upload files in raw_materials

Here we fill the local folders with the collected material. In the case of CMM, we got a tar.gz file containing the various versions of the software, organized according to an ad-hoc versioning system. In the raw_materials folder we store also the paper presenting the software and the email that Giuseppe Attardi sent us along with them, and we commit all these new contents:

```
git add raw_materials
git commit -m "Added raw material"
git push
```

The resulting state of raw_materials is shown in Figure {fig:cmm_rawempty citation}.

Unpack the source code in the browsable_source directory

In order to get a browsable version of the source code, we decompress the .tar.gz archive into the browsable_source folder

```
tar -xzf raw_material/cmm.tgz -C browsable_source
```

and commit the changes as done for the raw_materials folder

```
git add browsable_source
git commit -m "Added browsable source"
git push
```

⁵<https://github.com/Unipisa/SWHAP-TEMPLATE>

⁶The repository can be either public or private according to the policy of the acquisition team.

⁷See [Appendix B](#) for a brief discussion on the convenience of working locally, rather than remotely via the web interface.

Unipisa / CMM-Workbench Private
generated from Unipisa/SWHAP-TEMPLATE

Watch 2 Star 0 Fork 0

Code Issues 1 Pull requests 0 Projects 0 Wiki Security Insights Settings

SHWAPPE Workbench of a Customisable Memory Manager Edit

software-heritage workbench swhappe Manage topics

2 commits 1 branch 8 releases 2 contributors View license

Tree: 8c00149c58 New pull request Create new file Upload files Find file Clone or download

scatenag Updated README Latest commit 8c00149 2 days ago

.github	Initial commit	2 days ago
.swhap	Updated README	2 days ago
browsable_source	Initial commit	2 days ago
metadata	Initial commit	2 days ago
raw_materials	Initial commit	2 days ago
source	Initial commit	2 days ago
README.md	Updated README	2 days ago

README.md

CMM Workbench

This **SWHAPPE** workbench is for the acquisition of the source code of CMM, a Customisable Memory Manager.

This workbench comes with a few folders to support the process.

Folder **raw materials** is for the original materials, as they have been found or submitted.

Folder **browsable source** is for a browsable version of the source code. Source files, with right extension, have to be accessible through the GitHub web interface, e.g., archives should be decompressed, images should be transcribed, etc.

Folder **source** is for the curated revision of the source code, as a base for the reconstruction of the development history as a git repository, i.e., a folder for each major version of the code.

Folder **metadata** holds various files with meta information used throughout the process.

Please refer to the **SWHAPPE** guidelines for greater details.

Figure 4: Instantiated workbench for CMM.

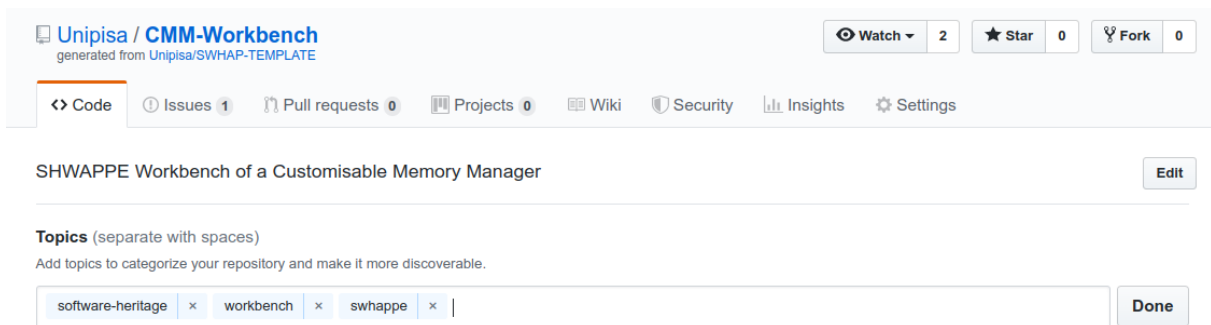


Figure 5: Tags for the workbench for CMM.

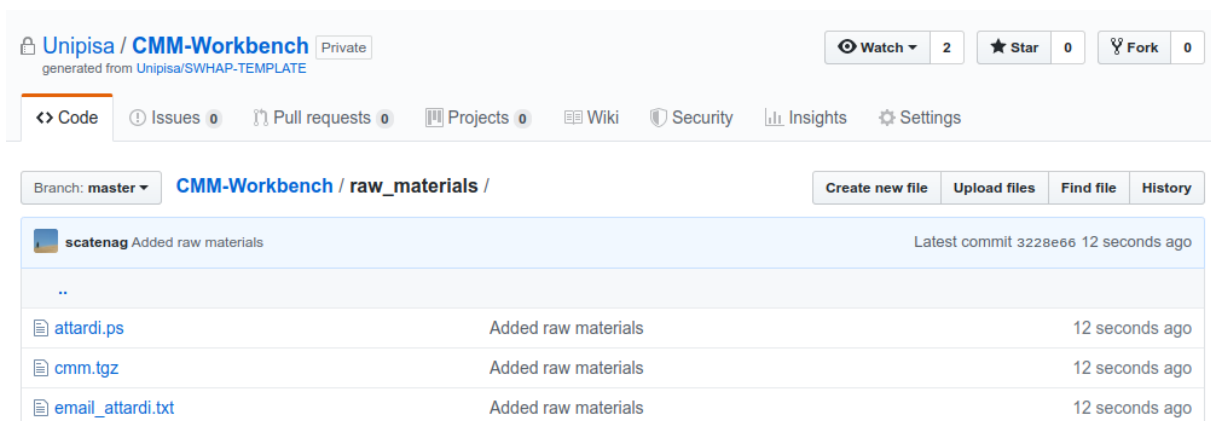


Figure 6: CMM raw materials on GitHub.

We can see the resulting state of the repository in Figure {fig:cmm_browseempty citation}.

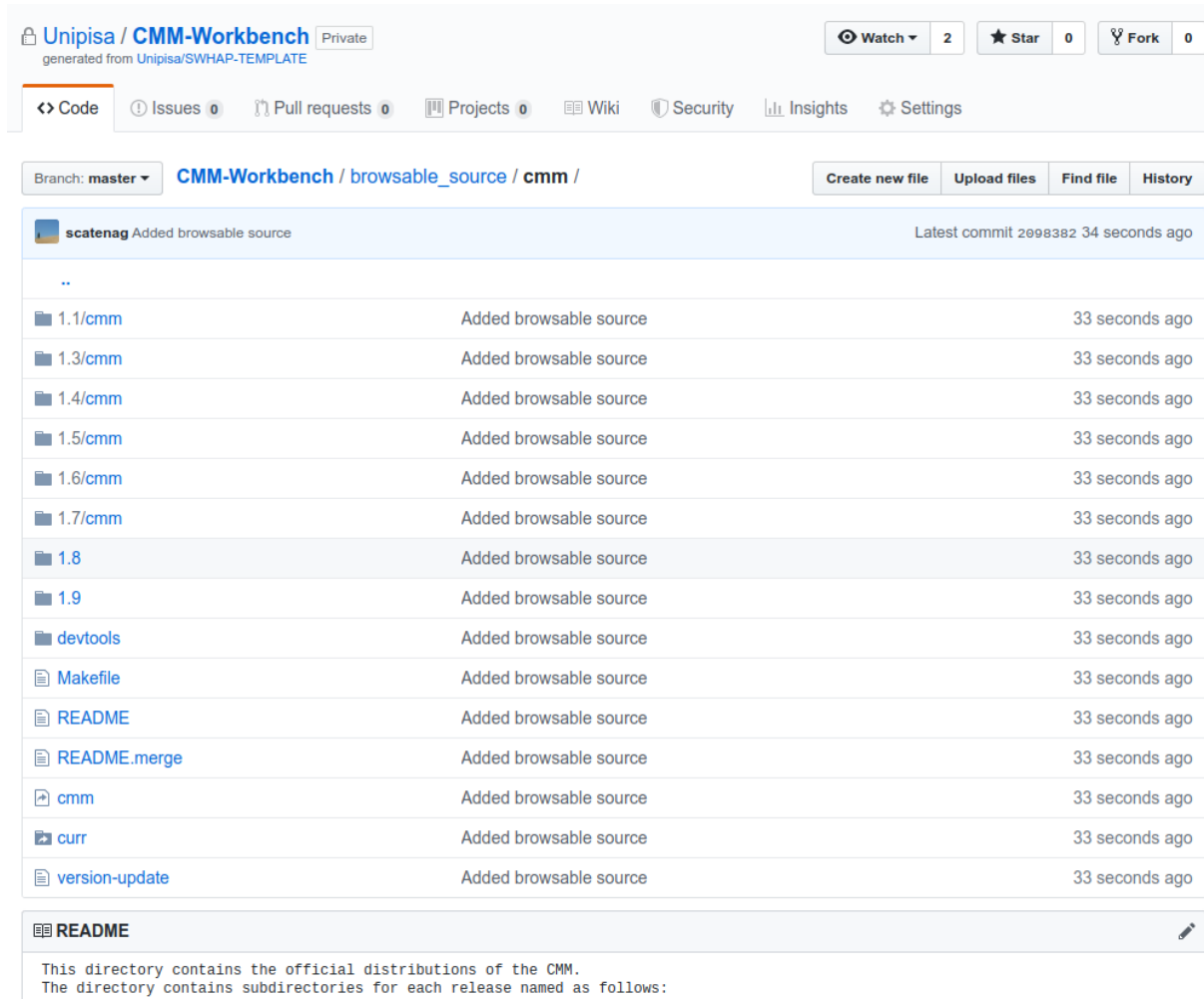


Figure 7: CMM browsable sources on GitHub.

Finally, in preparation for the next phase, curation, we copy the files contained in `browsable_source` into the `source` folder⁸.

```
cp -r browsable_source source
```

Again, we stage changes as in the previous two steps.

```
git add source
git commit -m "Added source"
git push
```

⁸Here shown with unix command line.

Create Depository

The Depository has been filled, hence we create the Depository as an orphan branch, i.e., with no references to the parent repository, using the checkout command:

```
git checkout --orphan Depository
```

As a result, we moved to the Depository branch. Here we modify the README (guidelines to fill the README file are given in the template) and remove the `source` and `metadata` folder, since they are not interesting for this area:

```
git rm -rf source metadata
```

We stage the last modifications and then push to the remote repository.

```
git add .
git commit -m "Added raw materials from main branch"
git push --mirror origin
```

We are almost ready to move the Depository to a new repository: before that, however, we have to create the new remote repository on GitHub (Figure {fig:cmm_depo_creatempty citation} shows how to do this using the web interface; here too one could use the GitHub API instead).

To facilitate the search of the created repository, we add the “software-heritage”, “depository” and “swhappe” tags (in the same way of what done for the workbench as shown in Figure {fig:workbench_tagsempy citation}).

Final depository

Finally, we can perform a push and fill the remote repository.

```
git push https://github.com/Unipisa/CMM-Depository.git +Depository:main
```

We can check the resulting repositories via the web interface (Figure {fig:cmm_reposempy citation}): CMM-Depository is now filled with the pushed materials.


The Depository branch is then removed from the Workbench, to avoid having multiple copies that may diverge. Should new materials became available, a new iteration of the process should start, re-initializing the Workbench with the information in the Depository.

```
git checkout main
git push --delete origin Depository
git branch -D Depository
```

Curate the code


Version History In this phase, the curation team should clean up the code and organize it in separate folders, one per version. In the case of CMM, the code is already structured this way, as shown in Figure {fig:cmm_reposempy citation}, so there is nothing to do.


Owner **Repository name ***

 **Unipisa** ▾ / **CMM-Depository** ✓

Great repository names are short and memorable. Need inspiration? How about **legendary-doodle?**

Description (optional)

☒  **Public**
Anyone can see this repository. You choose who can commit.

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

Skip this step if you're importing an existing repository.

☐ **Initialize this repository with a README**
This will let you immediately clone the repository to your computer.

Add .gitignore: **None** ▾ | Add a license: **None** ▾ ⓘ

Create repository

Figure 8: CMM-Depository creation.

Find a repository... Type: **All** ▾

CMM-Depository Private

● C++ Updated 16 hours ago

CMM-Workbench Private

● C++ Updated 16 hours ago

Figure 9: The CMM repositories at the end of the collect phase.

In order to support the (re-)creation of the development history of the original project, we prepare the `version_history.csv` file with the appropriate metadata (see Figure {fig:cmm_vers_histempty citation}).

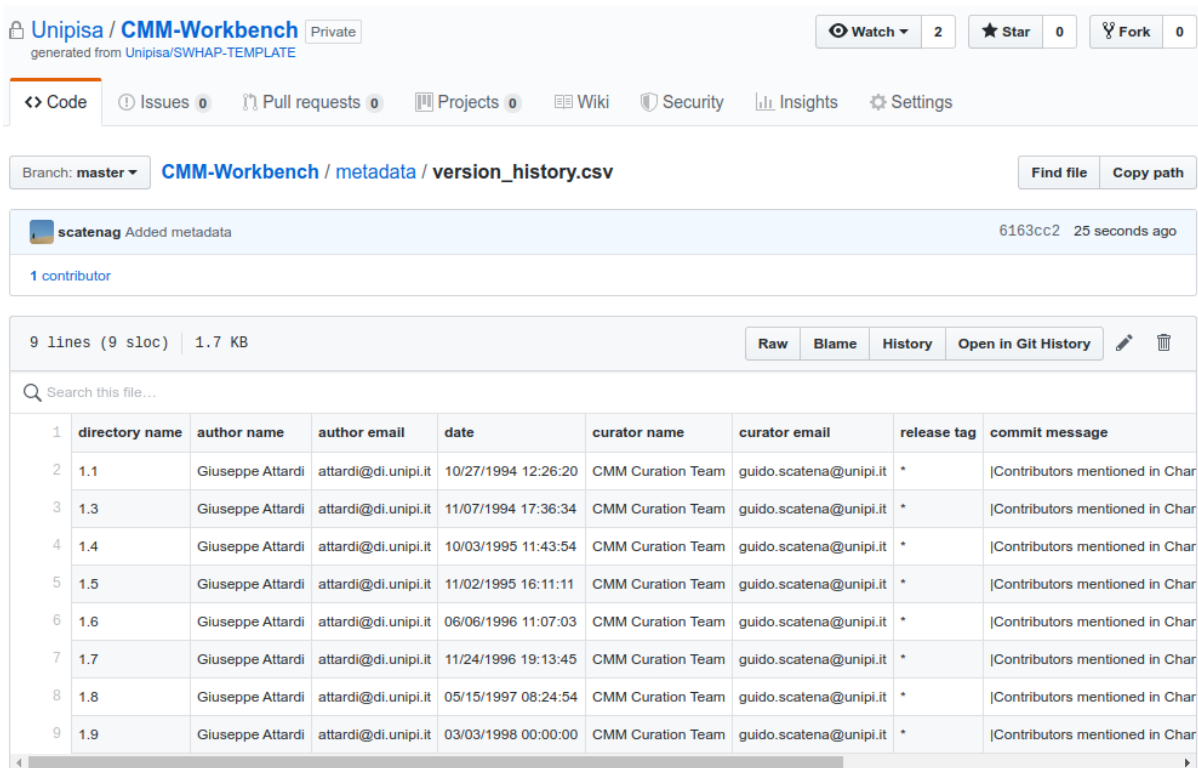


Figure 10: The version history for CMM

Codemeta Contextually we fill the `metadata/codemeta.json` template file (see Figure {fig:cmm_jsonempty citation}, left) with metadata according to CodeMeta guidelines obtaining what shown in (see Figure {fig:cmm_jsonempty citation}, right).

License To conclude the curation phase, we have to identify licensing information.

If we find a file specifying the licence in the source code, we have to copy its content in the `metadata/LICENCE` file. Otherwise, in the case there is no licensing file in the source and we obtained license information in other finds, we fill `metadata/LICENCE` according to the SPDX standard.

(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 specialised tool.

Manually We have to create a clean dedicated SourceCode branch

```
git checkout --orphan SourceCode
git rm -rf *
```

```

{
  "@context": "https://doi.org/10.5063/schema/codemeta-2.0",
  "@type": "SoftwareSourceCode",
  "identifier": "a unique identifier",
  "description": "a brief description of the software",
  "applicationCategory": "Type of software application, e.g. 'Game, Multimedia'",
  "name": "software name",
  "codeRepository": "URL to related resources",
  "license": "info about the license",
  "version": "X.X",
  "keywords": [
    "tag for the software",
    "other tag"
  ],
  "releaseNotes": "release notes and comments",
  "funding": "software funded by...(e.g. specific grant)",
  "runtimePlatform": "operating system",
  "softwareRequirements": [
    "dependencies and other requirements"
  ],
  "developmentStatus": "from https://www.repostatus.org/ e.g. Unsupported",
  "dateCreated": "YYYY-MM-DD",
  "datePublished": "YYYY-MM-DD",
  "relatedLink": [
    "https://...",
    "http://..."
  ],
  "programmingLanguage": "main programming language",
  "isPartOf": "maybe the software is part of some project",
  "referencePublication": [
    {
      "@type": "ScholarlyArticle",
      "identifier": "https://doi.org/xx.xxxx/xxxx.xxxx.xxxx",
      "name": "title of publication"
    }
  ],
  "author": [
    {
      "@type": "Person",
      "givenName": "Given",
      "familyName": "Family",
      "affiliation": "affiliation at the time of development",
      "email": "given.family@gorg.com"
    }
  ],
  "contributor": [
    {
      "@type": "Person",
      "givenName": "Given",
      "familyName": "Family",
      "affiliation": "affiliation at the time of development",
      "email": "given.family@gorg.com"
    }
  ]
}

```

```

{
  "@context": "https://doi.org/10.5063/schema/codemeta-2.0",
  "@type": "SoftwareSourceCode",
  "identifier": "CMM",
  "description": "conservative garbage collector for C++ developed for PoSso",
  "name": "Customizable Memory Management",
  "codeRepository": "https://github.com/Unipisa/CMM",
  "applicationCategory": "Memory Management",
  "version": "1.9",
  "keywords": [
    "Garbage Collector",
    "Memory Management"
  ],
  "runtimePlatform": "SunOS 4.x, Solaris 2.x, Linux 1.x, 2.x, AIX (RS6000)",
  "softwareRequirements": [
    "g++"
  ],
  "developmentStatus": "Unsupported",
  "dateCreated": "1994-08-27",
  "datePublished": "1995-08-26",
  "programmingLanguage": "c++",
  "isPartOf": "PoSso (Polynomial System Solver)",
  "referencePublication": [
    {
      "@type": "ScholarlyArticle",
      "identifier": "https://doi.org/10.1006/jasco.1996.0013",
      "name": "Memory Mangement in the PoSso Solver"
    }
  ],
  "author": [
    {
      "@type": "Person",
      "givenName": "Giuseppe",
      "familyName": "Attardi",
      "affiliation": "Dipartimento di Informatica, Università di Pisa"
    },
    {
      "@type": "Person",
      "givenName": "Tito Flagella",
      "familyName": "Tito Flagella",
      "affiliation": "Dipartimento di Informatica, Università di Pisa"
    },
    {
      "@type": "Person",
      "givenName": "Pietro",
      "familyName": "Iglorio",
      "affiliation": "Dipartimento di Informatica, Università di Pisa"
    }
  ],
  "contributor": [
    {
      "@type": "Person",
      "givenName": "Carlo",
      "familyName": "Traverso",
      "affiliation": "Dipartimento di Informatica, Università di Pisa"
    }
  ]
}

```

Figure 11: CMM instantiation (right) of codemeta.json template (left)

Then, for every directory containing a version of the source code, in chronological order, we copy its contents from the main branch to the current branch, and commit it with the appropriate metadata.

For example, for the directory 1.9 of the CMM sources, here is how we copy the source contents into our branch:

```
git checkout main -- source/1.9
mv source/1.9/* .
rm -rf source
```

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'
```

We also need to add an annotated tag to this version. For version 1.9 of CMM, here is the command we used, you can adapt it to your needs:

```
git tag -a 1.9 -m "Version 1.9"
```

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

```
git rm -rf *
```

With DT2SG And here is an example using the DT2SG tool

```
dotnet ./DT2SG/DT2SG_app.dll
-r SWHAP-EXAMPLE/CMM-Workbench
/source/cmm/
-m SWHAP-EXAMPLE/CMM-Workbench
metadata/version_history.csv
```

As a result we will find in our local repository a new local branch containing the rebuilt version history, that is shown in Figure {fig:cmm_shempty citation}.

Create the final repository

We move back to the main branch using the checkout command, then remove raw_materials, browsable_source and source from it:

```
git rm -rf raw_materials browsable_source source
```

We now create the README.md file, add it and commit changes:

Unipisa / CMM Private

Watch 2 Star 0 Fork 0

Code Issues 0 Pull requests 0 Projects 0 Wiki Security Insights Settings

Branch: SourceCode

Commits on Sep 23, 2019

1.9 - ...	Giuseppe Attardi authored and scatenag committed on Mar 3, 1998	4d46ebd
Contributors mentioned in Changelog : - Giuseppe Attardi @attardi - Tito Flagella @tflagella - Pietro Iglio		
1.8 - ...	Giuseppe Attardi authored and scatenag committed on May 15, 1997	7332413
1.7 - ...	Giuseppe Attardi authored and scatenag committed on Nov 24, 1996	fbcc896
1.6 - ...	Giuseppe Attardi authored and scatenag committed on Jun 6, 1996	127f2b6
1.5 - ...	Giuseppe Attardi authored and scatenag committed on Nov 2, 1995	a5c7b67
1.4 - ...	Giuseppe Attardi authored and scatenag committed on Oct 3, 1995	fa9db0d
Contributors mentioned in Changelog : - Giuseppe Attardi @attardi - Tito Flagella @tflagella		
1.3 - ...	Giuseppe Attardi authored and scatenag committed on Nov 7, 1994	d5fc834
Contributors mentioned in Changelog : - Giuseppe Attardi @attardi.		
1.1 - ...	Giuseppe Attardi authored and scatenag committed on Oct 27, 1994	93765ea

Figure 12: An excerpt of the synthetic history of CMM.

```
git add README.md
git commit -m "Final repository created"
```

Now we create the final remote repository, that we call “CMM”, see Figure {fig:create_finempty citation}, and we push the relevant branches (and tags) to it.

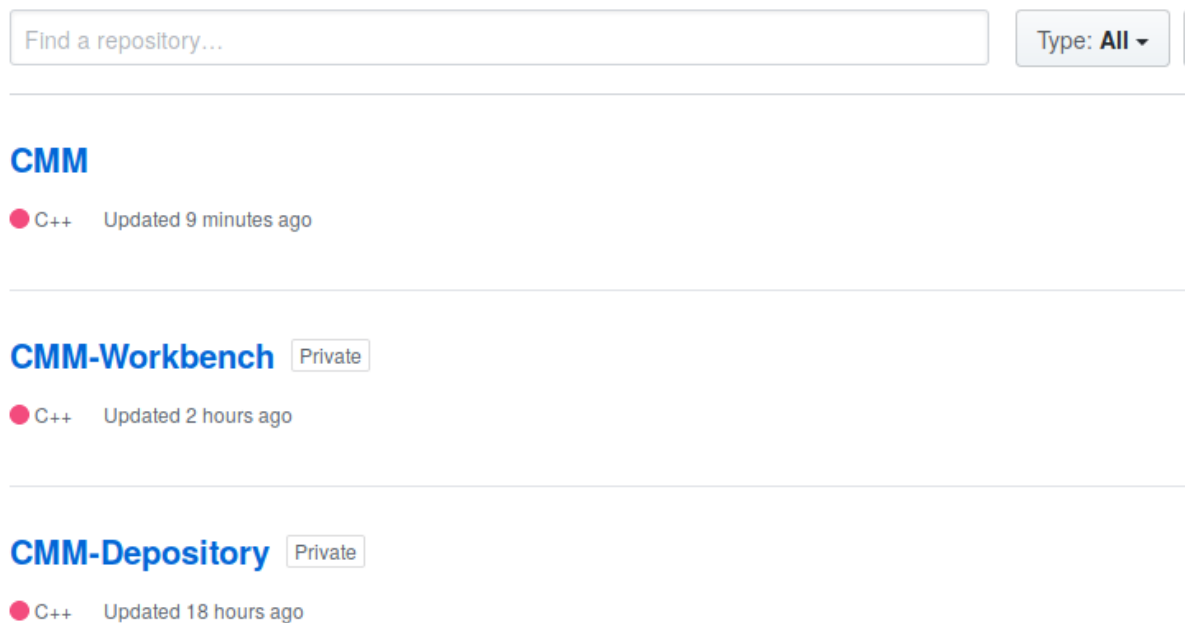


Figure 13: The creation of the final repository.

```
git push --tags git@github.com:Unipisa/CMM.git +main:main +SourceCode:SourceCode
```

To facilitate the search of the created repository, we add the “software-heritage”, “archive” and “swhappy” tags (in the same way of what done for the workbench as shown in Figure {fig:workbench_tagsempy citation}).

Figures {fig:cmm_finalempy citation}, {fig:cmm_depoempty citation}, {fig:cmm_wbempty citation} show the final result of CMM, their Depository and Workbench.

Publish the repositories and trigger Software Heritage acquisition

In order to publish the Depository and SourceCode repositories we have to set their visibility to “public”, either through GitHub web interface or using the GitHub API as follows:

```
curl -s -H 'Authorization: token $auth_token'
-H 'application/vnd.github.baptiste-preview+json'
--data '{"private": false}'
-X PATCH https://api.github.com/repos/$org/$repository_archive
```

Unipisa / CMM

Sponsor Watch 4 Star 0 Fork 0

Code Issues 0 Pull requests 0 Projects 0 Wiki Security Insights Settings

a Customisable Memory Manager Edit

software-heritage archive swhappy Manage topics

8 commits 2 branches 8 releases 1 contributor View license

Branch: master New pull request Create new file Upload files Find file Clone or download

scatenag Final repository created		Latest commit 819176c 2 days ago
.github	Initial commit	2 days ago
LICENSE	Final repository created	2 days ago
README.md	Updated README	2 days ago
codemeta.json	Final repository created	2 days ago

README.md

CMM

a Customisable Memory Manager

This repository contains the [CMM Development History](#).

The original finds are stored in the [Depository](#) containing the [raw materials](#) and the [browsable source](#).

[Information on the acquisition of this code](#) can be found in the [CMM-Workbench](#) repository.

This repository was created with the support of the [Software Heritage Acquisition Process Pisa Enactor](#).

Figure 14: The final CMM repository.

The screenshot shows the GitHub interface for the repository 'Unipisa / CMM-Depository'. At the top, there are navigation tabs: 'Code', 'Issues' (0), 'Pull requests' (0), 'Projects' (0), 'Wiki', 'Security', 'Insights', and 'Settings'. Below these, the repository name 'SHWAPPE Depository of a Customisable Memory Manager' is displayed, along with an 'Edit' button. A secondary navigation bar includes 'software-heritage', 'depository', 'shwappe', and 'Manage topics'. The repository statistics show 1 commit, 1 branch, 0 releases, and 1 contributor. Action buttons include 'Branch: master', 'New pull request', 'Create new file', 'Upload files', 'Find file', and 'Clone or download'. A list of recent commits is shown, with the latest commit 'scatenag Updated README' from 2 days ago. Below the commit list, the 'README.md' file is selected, displaying the repository's purpose and links to related resources.

Unipisa / **CMM-Depository** Watch 3 Star 0 Fork 0

<> Code Issues 0 Pull requests 0 Projects 0 Wiki Security Insights Settings

SHWAPPE Depository of a Customisable Memory Manager Edit

software-heritage depository shwappe Manage topics

1 commit 1 branch 0 releases 1 contributor

Branch: master New pull request Create new file Upload files Find file Clone or download

scatenag Updated README Latest commit d2ea39c 2 days ago

.github	Updated README	2 days ago
browsable_source/cmm	Updated README	2 days ago
raw_materials	Updated README	2 days ago
README.md	Updated README	2 days ago

README.md

CMM Depository

This is the depository for the acquisition of CMM, a Customisable Memory Manager.

This repository contains the [raw materials](#) and the [browsable source](#) of CMM.

[Information on the acquisition process](#) can be found in the [Workbench](#).

The resulting [development history](#) can be found in the [CMM repository](#).

This repository was created with the support of the [Software Heritage Acquisition Process Pisa Enactor](#).

Figure 15: The final CMM Depository.

Unipisa / **CMM-Workbench** Private
generated from Unipisa/SWHAP-TEMPLATE

Watch 2 Star 0 Fork 0

Code Issues 1 Pull requests 0 Projects 0 Wiki Security Insights Settings

SHWAPPE Workbench of a Customisable Memory Manager Edit

software-heritage workbench swhappe Manage topics

16 commits 1 branch 8 releases 2 contributors View license

Branch: master New pull request Create new file Upload files Find file Clone or download

scatenag	Update journal.md	Latest commit e13adf5 19 hours ago
.github	Initial commit	2 days ago
metadata	Update journal.md	19 hours ago
LICENSE	Final repository created	2 days ago
README.md	Workbench cleaned and README updated	2 days ago
codemeta.json	Final repository created	2 days ago

README.md

CMM Workbench

This **SWHAPPE** workbench is for the acquisition of the source code of CMM, a Customisable Memory Manager.

This repository was created with the support of the [Software Heritage Acquisition Process Pisa Enactor](#) and contains the [Information on the acquisition process](#).

The original finds are stored in the [Depositary](#) containing the [raw materials](#) and the [browsable source](#).

The resulting [development history](#) can be found in the [CMM repository](#).

Please refer to the [SWHAPPE](#) guidelines.

Figure 16: The final CMM Workbench.

where `$repository_archive` is CMM or CMM-Depository and `$auth_token` is the authorization token. As a result, the code is now publicly visible at

<https://github.com/Unipisa/CMM/>

Finally, we trigger the archival of this repository in Software Heritage, using the “save code now” functionality. This can be done using the web interface at <https://save.softwareheritage.org>, or by connecting to the API on the command line as follows:

```
curl -s -X POST https://archive.softwareheritage.org/api/1/origin/save/git/url/$repo_url
```

where `$repo_url` is <https://github.com/Unipisa/CMM/>.

A short time after (this may go up to a few hours for huge repositories), the archived software will be visible in the Software Heritage archive at

<https://archive.softwareheritage.org/browse/origin/https://github.com/Unipisa/CMM>

Fill the Workbench metadata

In order to preserve information about the curation process we have to fill the template files under the Workbench metadata. Starting from some template files (see Figure {fig:cmm-metadataempty citation}, left), we obtain what shown in Figure {fig:cmm-metadataempty citation}, right.

In particular we should create :

- a catalogue : `metatdata/catalogue.md`, where each item in the `raw_materials` should have a record describing its origin, the possible warehouse, their authors and collectors along with a description. The result of `tree -a` on `raw_materials` should be included;
- a journal : `metatdata/journal.md`, where each collect and curate action should be annotated;
- an actors registry: `metatdata/actors.md`, every person taking part in the process should be registered, with their roles, affiliations and contact information;
- a notepad : `metatdata/journal.md` where write possible information not covered by previous files.

Codemeta Best Practices for SWHAP

Each SWHAP workbench must include a `codemeta.json` file, providing machine-readable metadata about the curated software. 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/`)

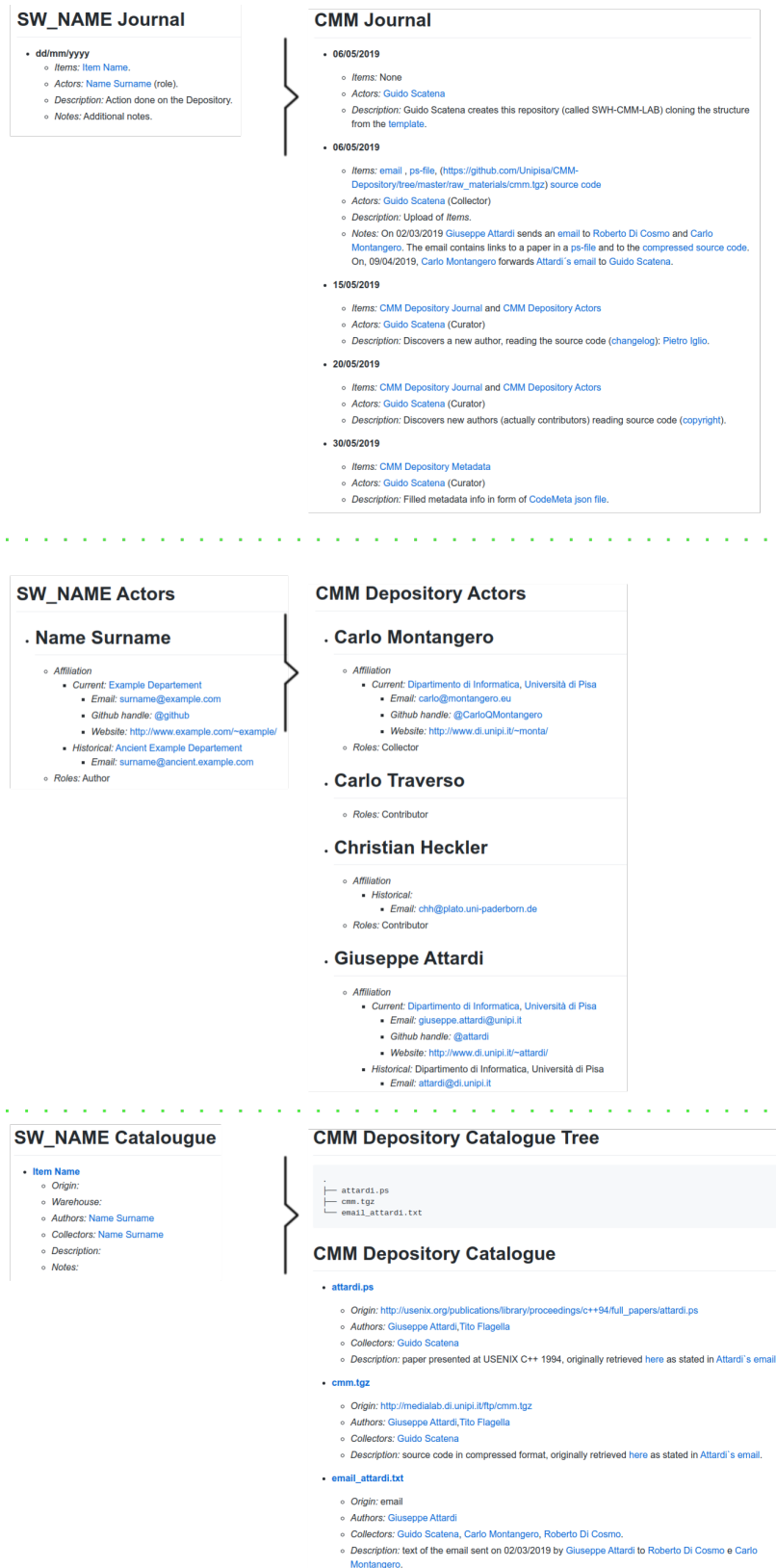


Figure 17: The CMM Metadata.

- 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).

3) Appendix A - Tools that can help

Here is a list of tools for code acquisition and curation that have been used during the initial experimentation of SWHAPPE:

- Used/suggested OCR:
 - Tesseract (<https://github.com/tesseract-ocr/>). It can be installed and used from command line. An API is also provided to use the OCR in a script.
 - OCR.space (<https://ocr.space/>). Online OCR and free API.
- Dedicated scripts:
 - DT2SG-Directory Tree 2 Synthetic Git (<https://github.com/Unipisa/SWHAP-DT2SG>). Creates the synthetic history of the software.
 - SWHAP-EXAMPLE(<https://github.com/Unipisa/SWHAP-EXAMPLE>)

Many other tools exist, and are currently under construction and will be loaded on the SWHAPPE repository on GitHub.

4) Appendix B - A few tips on Github

Git is a distributed version-control system for tracking changes in source code during software development. Here, we provide some references on *Git* and the GitHub platform.

For a review on GitHub key concepts, you can see the following glossary:

<https://help.github.com/en/articles/github-glossary>.

In order to fully exploit Github, you should install *Git* on your pc:

<https://git-scm.com/book/en/v2/Getting-Started-Installing-Git>.

This will allow you to use Git from command line. Even if it can be less intuitive, it's more powerful than working with the web interface: for instance, you can upload folders and files of any size, without the limitations of the latter. Furthermore, using Git commands allows for instantiating the process on any Git supported platform. For a review of the commands, please check the manual: <https://git-scm.com/docs>.

As an alternative, if you're using a Mac or Windows, you can download Github Desktop, which provides a comfortable GUI: <https://desktop.github.com/>.

For more information about the commit mechanism and how to see the log of changes, please see the following link: <https://git-scm.com/book/en/v2/Git-Basics-Viewing-the-Commit-History>.

To implement the process and separate areas, we chose to create two different branches (Depository and SourceCode) and get the corresponding repositories from them. Each branch has an independent commit history, thus the history of Depository and SourceCode is kept clean and easy to consult. Here is a discussion on how to see the branch history: <https://stackoverflow.com/questions/16974204/how-to-get-commit-history-for-just-one-branch>.