Paula Luput (pluput)
Carina Gordon (carinago)

SerenityOS

Selected project

The name of the open-source project we chose is called SerenityOS. It is a Unix-like operating system with an old school look, written as a love letter to the 90's user interfaces. It includes a browser, multiple classic games (like chess, Minesweeper, Solitaire), a coding IDE, a filesystem, and many other features. SerenityOS is open to anyone who misses the old-school 90's computer aesthetic. It is considered a not-for-profit project.

Social Good indication

SerenityOS does not contribute to Social Good.

Project context

The SerenityOS project was created in 2018 by Andreas Kling, a Swedish programmer, as a way to help his recovery from addiction. The name 'Serenity' came from the popular *Serenity Prayer*, which is commonly used amongst people who are trying to overcome alcohol/drug abuse and speaks on wanting serenity for the unchangeable and wisdom to change what should be changed. Kling launched the open-source project on GitHub and started to grow it alongside a few other developers who were working on Serenity as a hobby. There are now nine main maintainers of the project, Kling included, and they are in charge of accepting, rejecting, and adding comments to PR's. They are all fairly active on Discord, especially Kling because he is a full-time worker for the project and is able to do that through community donations. SerenityOS has grown to include a fully functional browser, multiple games and demos, support for many common and uncommon file formats, and more. The project boasts its 53 authors who have done over 100 commits and 813 contributors total. There are also almost 7.5k members on the project's Discord and 2.5k forks of the Github repository have been made. Clearly there are many people excited about and interested in the development of SerenityOS. Its target audience is its developers, so its goal is *not* to gain the largest possible userbase. It also is not intended to be sold for a profit, but is for the pure passion of developers.

Project Governance

Communication is semi-casual, and is mostly conducted on Discord. Discord is the preferred method of communication and is very active. It's divided into different channels for the different parts of the project (i.e. "browser", "build-problems", "ui-development", etc). Project maintainers encourage asking questions on Discord and their members usually respond quickly. The main usage of their Discord is for contributors to ask and answer questions. There is also a bot that automatically sends out an alert every time a PR is sent in. However, contributors can send their own message in the "code-review" channel and announce their PR if they want attention on it right away. Their Discord seems semi-casual because people have a lot of autonomy over what they say, except for some fairly obvious rules: "be positive", "don't complain", "don't solicit", etc.

Contributing to the project is also semi-formal. Contributions are welcome by everyone and there is a set process in place to help organize things. This process is specified through their contributing guide linked on the project's landing page. It encourages new contributors to start off with small tasks instead of attempting to add big changes to the project. It also includes an issue policy (for submitting bug issues), a human language policy (for commits and comments), a code submission policy (for writing code), and a pull request Q&A. All of these policies are clearly specified. However, there isn't an established task assignment process which is why we consider contributing to be semi-formal.

When attempting PR acceptance, the code submission process and the pull request Q&A are very important. All pull requests need to specifically be about the issue the programmer is trying to solve, so code files outside of the scope should not be touched. Before committing, contributors should run commit hooks to ensure their code will pass the linting tests on the automated Continuous Integration (CI) GitHub tests. Then, after submitting a pull request, 13 CI tests are run on Github. There are 4 possible phases a PR may be in before getting accepted: "pr-waiting-for-author" when the PR is blocked by code changes from author; "pr-has-conflicts" when the PR has merge conflicts and needs to be rebased; "pr-needs-review" when the PR needs review from a maintainer; and "pr-maintainer-approved-but-awaiting-ci" when the PR has been approved and can be merged after CI has passed. These 4 phases are different labels that project maintainers will add to PRs. Project maintainers are also in charge of accepting, rejecting and leaving comments in code. If a PR hasn't been touched in 21 days, maintainers will label it stale and then close it after 7 more days. However, abandoned PR's could still be used by maintainers if they are considered good.

Task description

Due to an unpreventable event, we had to switch tasks chosen in HW6a. Another contributor implemented the API endpoint that we were planning to implement (POST /session/{session id}/window), so we chose another open API endpoint instead (POST /session/{session id}/print). This task involved implementing an API endpoint that prints out the current page by rendering a PDF representation of the page document.

We started by splitting this task up into 27 smaller subtasks, which we obtained from the spec on the print API endpoint. We implemented 22/27 subtasks by researching tasks in the spec and by digging through existing source code to try and find similar code to model off of. At a high level, we gathered all of the relevant variables needed for printing the page (i.e. page orientation, page width, margin, scale, etc) and performed error checking to ensure that the variables were formatted correctly. Lastly, we did a lot of external research on the internet when debugging and understanding the context of our task.

Submitted artifacts

Link to pull request on github: https://github.com/SerenityOS/serenity/pull/16269/files. This is our submitted pull request from our forked branch to the main project repository. We edited 4 files with our changes shown under the "Files changed" tab. We have a very brief description of our pull request in the first comment under the "Conversation" tab.

QA strategy

The dominant QA strategy we performed was pair programming. The main reasons we used pair programming were to ensure mutual understanding, reduce defects, and encourage collaboration. When we started this project, we were both very confused by a lot of the source code and how to implement the tasks. So, by pair programming in the beginning stages of brainstorming implementation ideas, we were able to bounce ideas off each other to try and find the optimal strategy. We would also ask each other for clarification on the other's ideas, forcing each of us to justify our approaches. This led to more thoughtful solutions. Also, by asking and answering each other's questions, we were able to ensure that we were on the same page through the development process. This was very valuable when working with a foreign project, especially since our closest resource for questions was each other. Once we began coding, the navigator was able to identify problems while the driver wrote code. Again, this led to more accountability on our code. When the driver codes, the navigator is able to devote all of their efforts to scrutinizing the code and making suggestions to improve it. This felt like we were conducting a constant code review and, because of this, we were able to write more quality code through the efforts of pair programming. We also feel that pair programming makes for a more enjoyable experience. If we did not employ it, each task likely would have felt more intimidating and difficult.

The second most dominant QA strategy we used was following the coding standards within the project. First, we followed the specification given for our task. By strictly adhering to the functional requirements for the issue (the API endpoint for print page) we were able to feel more confident in our solution. Although the specification left a lot of room for interpretation, it still gave an overall guideline for us to follow. This gave us the opportunity to be accountable and make sure we were on the right track. Second, we followed the design patterns we observed in the source code. By basing our code off of the existing code, we were able to clarify general patterns like how certain variables should be fetched or how error handling should be implemented. This increased our confidence that we were writing quality code since our implementation was similar to existing code. Through comparing the source code with our code, we were able to tailor our implementation and make it more likely to be accepted. Third, we followed the general coding standards. This included how variables should be named and how scope should be set. Again, this helped guide us in the right direction. Lastly, we also followed the guidelines that were related to code style. This allowed us to create code that would be easier to read and maintain, since its style was the same as the rest of the codebase.

The last QA strategies we used were provided by the Github project, like linting and automated CI tests. Before submitting our PR, we were able to download commit hooks which ran the linting tests against our changed code. This notified us of any errors we had by identifying the error-prone lines, making it helpful for us to debug. After submitting the PR, several other CI tests were run automatically, including build tests and fuzzing tests. Although not all of these CI tests gave helpful output, they were still useful for seeing what we were doing right/wrong based on the test names.

QA evidence

Linter

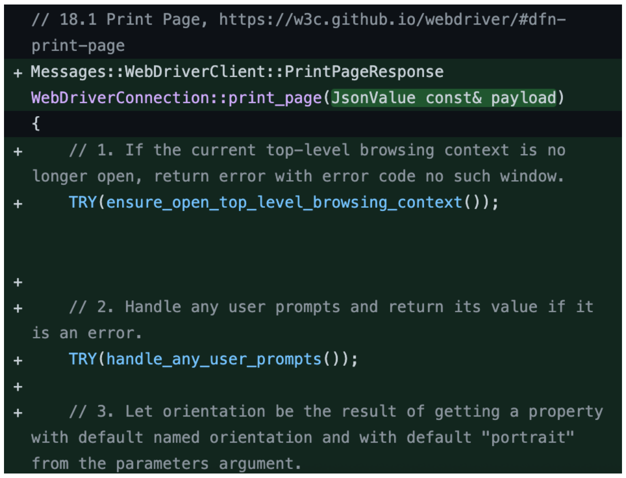
We ran the programs commit hooks before and after creating our commit. These commit hooks lint our commit message and the changes made in our commit. We ran this before we opened up our pull request and were able to fix a few programming and stylistic errors before sending in our commit. The linting checks are shown in our pull request at: https://github.com/SerenityOS/serenity/actions/runs/3625353866/jobs/6113311763.

CI Tests

Our pull request automatically sent our code through their main tests that build and run serenity as well as using Lagom to fuzz parts of SerenityOS's code base. These tests can be found at: https://github.com/SerenityOS/serenity/actions/runs/3625353878. The main test runs allow programmers to check if their code changes pass their tests for successfully building and running different parts of Serenity. Then their fuzzing creates interesting test cases based on variations of SerenityOS's code and used to check for defects in our code.

Following Code Structure

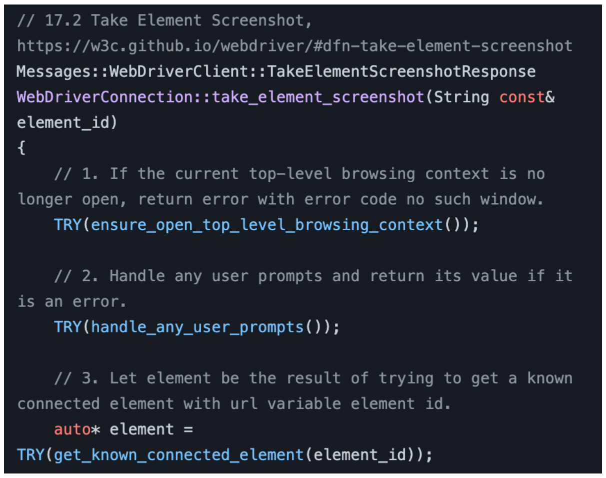
To ensure maintainability of our code, we wanted to follow the given coding standards and guidelines. One way we did this is by following the given specification for the print page endpoint, specified at https://w3c.github.io/webdriver/#print-page. The screenshot below shows how we added comments for each step and put our implementation below it. Another way we followed the established code structure was by following the code style guidelines (https://github.com/SerenityOS/serenity/blob/master/Documentation/CodingStyle.md) and the general, non-style related, coding standards (https://github.com/SerenityOS/serenity/blob/master/CONTRIBUTING.md#code-submission-policy). This was done through actions such as setting our variable names to snake cases. The last way we did this is by following the code structure from other similar parts of the codebase. Below is an example of how we followed this through things like comment structure and spacing/indentation style.



```
  // 18.1 Print Page, https://w3c.github.io/webdriver/#dfn-
  print-page
+ Messages::WebDriverClient::PrintPageResponse
  WebDriverConnection::print_page(JsonValue const& payload)
  {
+     // 1. If the current top-level browsing context is no
  longer open, return error with error code no such window.
+     TRY(ensure_open_top_level_browsing_context());


+
+     // 2. Handle any user prompts and return its value if it
  is an error.
+     TRY(handle_any_user_prompts());
+
+     // 3. Let orientation be the result of getting a property
  with default named orientation and with default "portrait"
  from the parameters argument.
```

Our implementation



```
// 17.2 Take Element Screenshot,
https://w3c.github.io/webdriver/#dfn-take-element-screenshot
Messages::WebDriverClient::TakeElementScreenshotResponse
WebDriverConnection::take_element_screenshot(String const&
element_id)
{
    // 1. If the current top-level browsing context is no
longer open, return error with error code no such window.
    TRY(ensure_open_top_level_browsing_context());

    // 2. Handle any user prompts and return its value if it
is an error.
    TRY(handle_any_user_prompts());

    // 3. Let element be the result of trying to get a known
connected element with url variable element id.
    auto* element =
TRY(get_known_connected_element(element_id));
```

Implementation we used as reference for coding style

Plan updates

Activity	Expected Duration	Actual Duration	Justification
Build and run SerenityOS locally	4 hrs (per person)	5 hrs (per person)	Our estimated and actual time spent are almost the same amount because we assumed beforehand that we would run into some build issues. However, we still had unexpected issues emerge. For example, one of us built and ran the program smoothly while the other kept stumbling across strange issues. Likewise, the actual successful build time was much longer than we anticipated, taking almost an hour total.
Read general project documentation	1 hr (together)	2 hrs (one person)	We had only one person read through the documentation and then summarize the most important parts for the other for efficiency. The actual time spent was double our time estimate because of the large amount of documentation there was to look through.
Read testing and requirements	1 hr (together)	2 hrs (one person)	We also assigned only one person to look through the project's testing organization and requirements. However, the large variety of testing to look over felt very overwhelming to try and understand. We decided to stop reading through everything after 2 hours, hoping it would make more sense to us once we got to testing.
Ask questions: If confused from previous tasks	1-2 hrs (one person)	None	We had no pressing questions about the project before coding, so we asked none.
Implement 1st task: Understand expected behavior	1 hr (together)	None	We found that there was no way for us to see how the API endpoints were working or visually changing something on the VM, so this step was skipped.
Implement first task: Do research on APIs	0.5 hrs (together)	0.5 hrs (per person)	This step was left for each person to do on their own so that we each would feel confident about how API's worked. We both still felt a little lost on how the API endpoints worked in the code but we had enough knowledge to feel okay with starting to code.
Implement first task: Look over similarly implemented endpoints	0.5 hrs (together)	1.25 hrs (together)	We were surprised at how complicated it was to understand the implemented examples. We tried to understand them as best as we could and then stopped after it became too complicated. We decided it was not worth our time to try and understand the functions that we knew we had to use regardless.
Implement first task: Brainstorm	1 hr (together)	10.25 hrs (together)	This step took over 10 times longer than our estimate because of unanticipated and anticipated issues. First, we struggled to simply open up the code to edit in XCode. After 2 hours of struggling, we

and design implementation together			decided to do pair programming with the driver who was able to open the code first. Then, after an hour of brainstorming, we saw that a PR was made that had almost completely implemented the initial API endpoint task we wanted to do. This was anticipated; we had our other tasks to choose from as backups. However, we both felt that our initial task was the easiest one for us to start with. Therefore, we decided to still work on an API endpoint, just a different one. We chose another one from the master list of API endpoints awaiting implementation. Then, we found out someone who implemented everything but the main function for the new task we picked, so we updated our local code and worked on finishing the rest of the task.
Implement first task: Run unit tests	0.5 hrs (together)	3 hrs (together)	There were no additional tests needed to be created so we only used the tests automatically run on PRs. Verifying our commit and code to be sure it followed the guidelines took a little more time than we assumed it would.
Implement first task: Debug implementation	1 hr or 3 hrs (together)	3.5 hrs (together)	We didn't get to debug the code as much as we wanted because we wanted to allocate enough time to write the report. The only tweaks we were able to do was fix a few errors reported after test runs and successfully squash our commits into one.
Implement second and third task	15-21 hrs (together)	None	Enough time had already been spent on this project; there was no need or time to implement our other tasks.
Writing Report	6 hrs (together)	14.5 hrs (together)	We spent 14.5 hours total with one of us spending 10 hours, the other spending 12 hours, totaling in 22 hours of work. The report ended up being a little more time consuming than we imagined and there was a lot of stress for it to be well-written which resulted in more time spent.

Our Experiences and Recommendations

There were a lot of challenges that we encountered while working on this project as neither of us had contributed to an open source project before. We struggled a lot with selecting our project and selecting which tasks to work on. Also, neither of us had worked on a project with as minimal guidance as this one. In our school and internship projects, we had many resources where we could get help and had specific people directing us on what to do. For this project, our only resources were each other, Discord, the source code, and Github.

Culture and Communication

When attempting to implement our first task, the print page API endpoint, we spent a few hours creating the stub for the endpoint. However, once we got to the main function we needed to write, we saw on GitHub that someone else had already pushed the same stub changes we had already made. All that was left to do for the print page API endpoint was to implement the main function, which we really wanted to undertake, so we decided to send out a message on Discord. Looking carefully at how the programmer who pushed the stub conversed on Discord, we let the community know that we planned on implementing the main function of the print page API endpoint, thus completing the endpoint. The conversations held on Discord felt very informal and friendly, even though there were many rules on what was appropriate to say. They wanted to limit extra noise being made, so we were hesitant to converse too much on Discord. Yet, when we stumbled upon a confusing issue, we tried to ask on Discord to see what other programmers recommended. Although we didn't receive any response for that specific question, in general, programmers always seemed to be willing to help out each other if they did have any insight.

The comments left under our pull request also gave us an insight on how the community communicates. We received three comments from two project maintainers and one automatic bot. Whenever we made changes to our pull request, a project maintainer would leave a comment or add a label relating to the phase of the PR within a few days. This demonstrated their dedication to the project but also their eagerness to support the community. The comments left were friendly and welcoming, which was a pleasant surprise. For example, the first comment we received was a welcome message as well as a friendly reminder on how to clean up our commits so that it's maintainable and readable for future contributions.

We appreciated working with a helpful and friendly community because it made us feel less intimidated to ask questions to submit a PR. Because the endeavor of contributing to an open source project was already intimidating, being a part of this community definitely improved our overall experience. Their comments and advice also helped us follow the same PR structure other programmers had. Even though we carefully read through the documentation on PR guidelines, there seemed to still be a few unspoken rules that the project maintainers wanted everyone to follow for better maintenance in the future, so we appreciated the guidance.

Challenges

The first battle we faced was understanding the project. At a superficial level, our project seemed simple: create a user interface for an operating system that had a vintage aesthetic. Yet when we actually looked at what this project was made of, we realized how complicated everything really was. We first struggled with selecting tasks that would be reasonable for us to implement. But reading into the tasks and the source code only made us more confused. This was an unexpected challenge for us. We knew that working on a foreign project would be intimidating, but neither of us expected to be so lost. We realized how much we'd need to learn to adequately differentiate the difficulty between tasks since we had no idea how to quantify the effort or time each task would require. We also were aware of the fact that SWE's are infamously bad at estimating project scope or duration. So, these factors combined with the fact that neither of us had the time to completely, thoroughly understand the project meant that we had to make a leap of faith. We knew we couldn't expect to fully understand our tasks before choosing them, otherwise we would get too overwhelmed. Thankfully, the first task we attempted to implement came with a specification (a spec that contained every API endpoint), and we used this to aid our understanding.

The second battle was implementing our task. Although we were guided by the task's spec, there was still a lot of research we needed to do. We did internal research by reading through source code and discord, and external research by clarifying concepts and terms on the internet. The spec was also vague at times, and we weren't always sure if we were interpreting everything correctly. In the spec, each API endpoint was divided into several sub-tasks to complete in order to implement the overall API endpoint. We attempted to seek clarification by searching through other API endpoints in the spec to see if any of our endpoints' sub-tasks were repeated elsewhere. We successfully found them a few times and modeled our code after that, but for the other subtasks, we made changes based on what we found online. When we sought out answers online, we struggled with determining if the general case discussed would be compatible with the specific task we were working on. We often assumed that the online solution would work, and rely on the future reviewers of our PR to address our errors otherwise. This was another unexpected aspect of the assignment because in school projects, if our code was wrong, we would fail our given tests. But in this situation, if our code was wrong, we could trust that someone would help us tweak our code to correct it. This was somewhat comforting and removed some of the pressure we felt about writing "perfect" code from the get go.

The last battle was contributing to the project and trying to get our code accepted. Once we felt somewhat confident in our implementation, we needed to get approval from the project maintainers. Our first step was running the lintings tests on our changed code, which helped us spot and correct errors. Then, we submitted a pull request, where automated CI tests ran. We ended up failing a lot of the CI tests, but were not sure why because most of the automated tests didn't output any error logs. This felt unexpectedly similar to the frustration of failing private autograder tests in school. Another contributing challenge we faced related to a comment written by a project maintainer recommending that we squash our commits. Although both of us have experience with git, and there are tons of resources about git online, this task turned out to be unexpectedly difficult. It seemed like every solution we found online was somehow not applicable to our specific problem. Although we eventually figured it out, we both felt frustrated by how much time we spent on what we thought would be a non-trivial task.

Reflection

Our biggest challenge for this project was underestimating how many difficulties we would face and how much time we'd spend overcoming them. Even though we started the project early and gave ourselves a lot of space to make mistakes, we still did not manage to fully implement the print page API endpoint. Different tasks like building and running or not being able to claim tasks made it difficult for us to finish things faster. However, a lot of our difficulties came from coding an unfamiliar task and understanding a completely foreign project. We had both experienced a summer internship where the codebase was huge and difficult to fully understand, but we had mentors at the time to walk us through different tasks. With this project, we had no assigned mentor that would be there to guide us and answer any question we had. Instead, there seemed to be good communication standards established between programmers and project maintainers. Our fault was that we were too nervous about saying something ignorant or unnecessary to take advantage of their communication system.

On the other hand, our system of meeting up almost every day and pair programming wherever we needed to code worked in our favor. Since we met up often, we were able to stay on track and communicate regularly about what our next steps would be. This allowed us to be more flexible with our timeline, like if we needed to spend more time on a task or change what our goals were. Meeting often also forced us to be in constant communication with each other, so we never felt unaware of what the other person was doing. Likewise, when we did pair programming, we were able to constantly check that we were on the same page. Our plan of action was clear for both of us at all times and we were able to work through issues we had together. Having one person constantly reviewing the other's work was also useful for both to feel more confident in what code we were implementing. Also, we were able to help each other understand certain details that the other didn't fully understand. That way we were both on the same page throughout the project and then we could both agree or disagree about whether what we were coding was right.

There were a few actions we wish we had done differently during this project in order to improve our productivity. The first action we would have changed is communicating more with the contributors on Discord. We could have engaged more with the friendly community and figured out a few things we were struggling with if we were not nervous to do so. This possibly could have helped us successfully finish our first task. Another thing we would have changed was our metric for success in the planning process. In the beginning, we were very ambitious and were striving for success based on how many tasks (or GitHub issues) we could complete. However, as we worked our way through the first task, we altered our metric for success to be how many subtasks (within the first task only) we could complete. Our initial success metric put unnecessary pressure on us, making us feel behind and very unsuccessful throughout most of the project. We could have prevented this stress by simply creating smaller milestones for us to accomplish. That way we could have considered smaller aspects like achieving a build, running our project, or implementing subtasks of our larger task, as successes.

Advice for future students

Paula: “Starting early but also *meeting often* was a huge support for us. We were able to adjust our goals and plans a lot more easily.”

Carina: “Start ASAP since many unpredictable obstacles may come up.”

**We are willing to let future students see our materials.