Programming in the Large (CSSE2002)

Assignment 1 — Semester 1, 2024

School of EECS

The University of Queensland

Due March 25th 13:00 AEST

One must learn by doing the thing; for though you think
you know it, you have no certainty, until you try.
— Sophocles

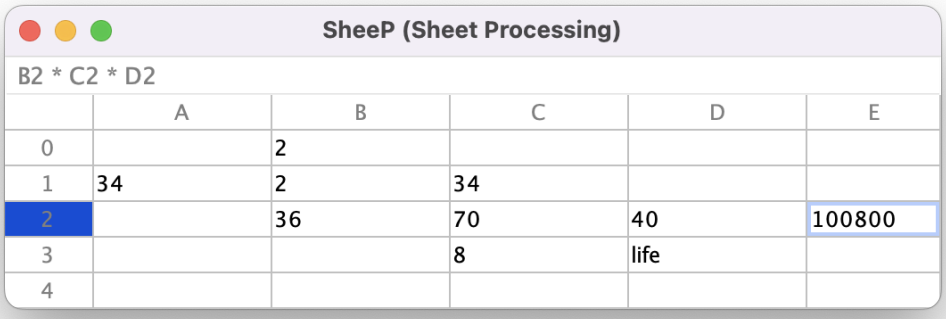
 Revision 1.0.0

Overview This assignment provides practical experience developing a Java project based on a supplied specification. The specification is provided in the form of JavaDocs, which describe the classes and interfaces that your assignment must implement. Additionally, you must develop JUnit tests for a subset of these classes.[1]

You will be evaluated on your ability to;

- implement a program that complies with the specification,
- develop JUnit tests that can identify bugs in class implementations,
- and write code that conforms to the style conventions of the course.

Task Spreadsheet applications are powerful tools that combine data and formulae to perform calculations. In this assignment, you will be implementing SheeP (Sheet Processor), a spreadsheet application. SheeP is similar to Google Sheets or Microsoft Excel, it consists of a grid of cells each of which contains either data or a formula. The formulae can reference other cells in the grid to use their values. Formulae are evaluated to produce a value for the cell. A cell is updated whenever the data or formulae in any cell it references changes.



Common Mistakes Please carefully read Appendix A. It outlines common and critical mistakes which you must avoid to prevent a loss of marks. If at any point you are even slightly unsure, please check as soon as possible with course staff.

[1] Although you are encouraged to write tests for all your classes.

Plagiarism All work on this assignment is to be your own individual work. Code supplied by course staff (from this semester) is acceptable, but must be clearly acknowledged. Code generated by third-party tools is also acceptable, but must also be clearly acknowledged, see *Generative Artificial Intelligence* below. You must be familiar with the school policy on plagiarism:

`https://uq.mu/rl553`

If you have questions about what is acceptable, please ask course staff.

Generative Artificial Intelligence You are strongly encouraged to *not* use generative artificial intelligence (AI) tools to develop your assignment. This is a learning exercise and you will harm your learning if you use AI tools inappropriately. Remember, you will be required to write code, by hand, in the final exam. If you do use AI tools, you must clearly acknowledge this in your submission. See Appendix C for details on how to acknowledge the use of generative AI tools. Even if acknowledged, you will need to be able to explain any code you submit.

Interviews In order to maintain assessment integrity and in accordance with section 5.4 of the course profile, you may be asked by the course coordinator via email to attend an interview to evaluate genuine authorship your assignment. Please refer to the course profile for further details.

Specification

The specification document is provided in the form of JavaDocs.

- Implement the classes and interfaces *exactly* as described in the JavaDocs.
- Read the JavaDocs carefully and understand the specification *before* programming.
- Do not change the public specification in any way, including changing the names of or adding additional public classes, interfaces, methods, or fields.
- You are encouraged to add additional *private* members, classes, or interfaces as you see fit.

To view the Javadoc specification, visit the below URL in your web browser:

`https://csse2002.uqcloud.net/assessment/ass1/docs/`

Getting Started

To get started, download the provided code from Blackboard. This zip archive includes code for the GUI components. Extract the archive in a directory and open it with IntelliJ.

Tasks

1. Fully implement each of the classes and interfaces described in the Javadoc specification.
2. Write JUnit 4 tests for all the public behaviour of the following classes:
   - `CellLocation`
     (in a class called `CellLocationTest`)
   - `Reference`
     (in a class called `ReferenceTest`)

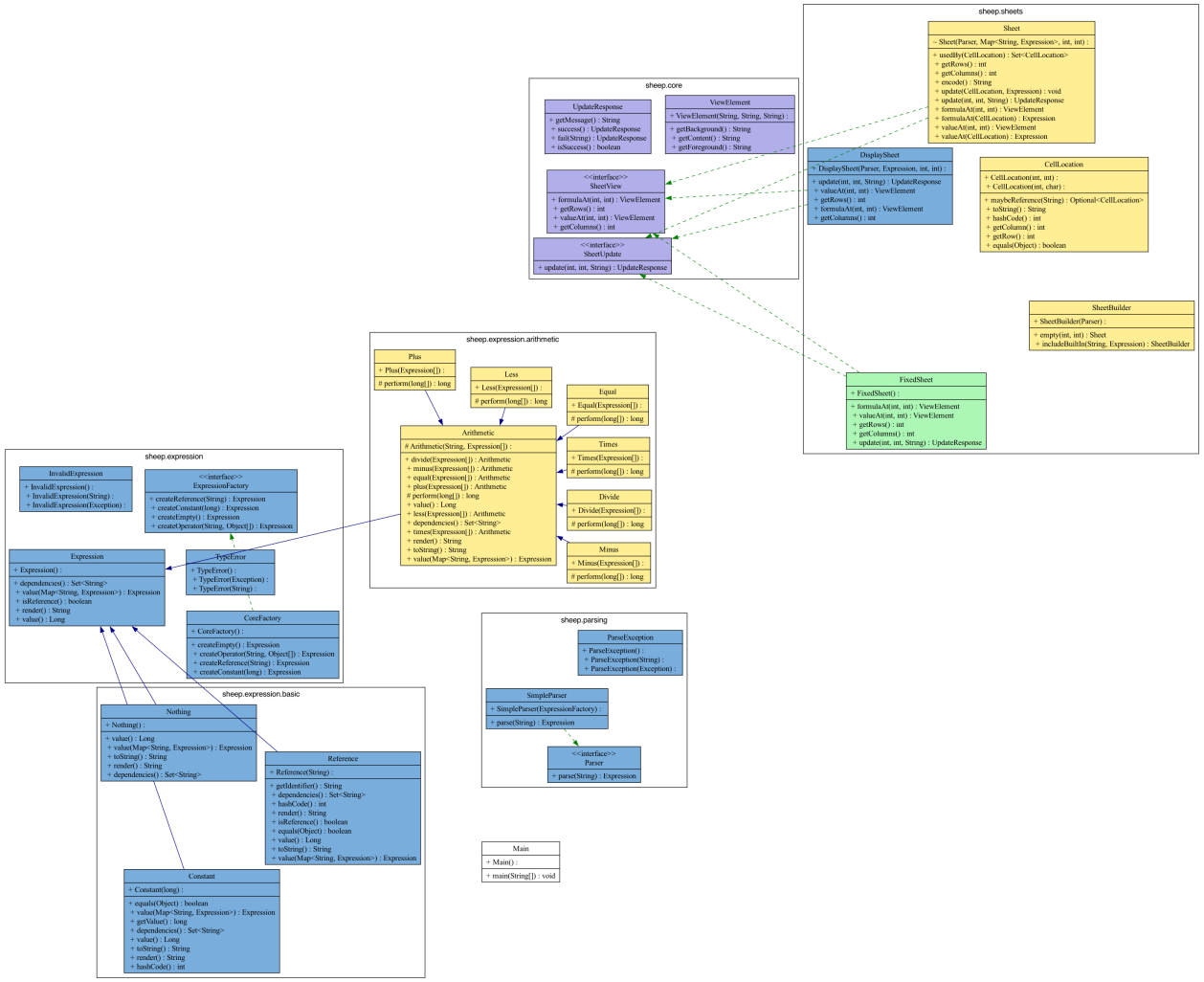


Figure 1: Class diagram of the specification for assignment 1.

Project Overview

sheep.core This package contains the interface between the model of a spreadsheet and the user interface.

Implementations of the `SheetView` interface tell the interface how it should render a spreadsheet and communicate this information via the `ViewElement` object.

Implementations of the `SheetUpdate` interface handle user updates to the spreadsheet and provide the result of the update via a `UpdateResponse` object.

sheep.sheets This package contains implementations of the `SheetView` and `SheetUpdate` interfaces and other supporting classes. Primarily it implements three different types of spreadsheets: `FixedSheet`, `DisplaySheet`, and `Sheet`.

sheep.expression Within a spreadsheet, the value at a particular cell is represented by an expression. This package stores the `Expression` interface that all expressions must implement.

Expressions are built via expression factories which implement the `ExpressionFactory` interface, e.g. `CoreFactory`.

This package also stores relevant exceptions.

sheep.expression.basic This package stores core expression implementations, namely, the empty cell, `Nothing`, a constant number value, `Constant`, and a reference to another cell, `Reference`.

sheep.expression.arithmetic Arithmetic expressions are contained in this package. All arithmetic expressions are subclasses of the base abstract class `Arithmetic`.

sheep.parsing A parser accepts string input and constructs an appropriate expression. For example, given the string "5", a parser would construct an instance of the `Constant` expression that represents 5.

All parsers must implement the `Parser` interface. If a parser cannot parse a string, a `ParseException` is thrown.

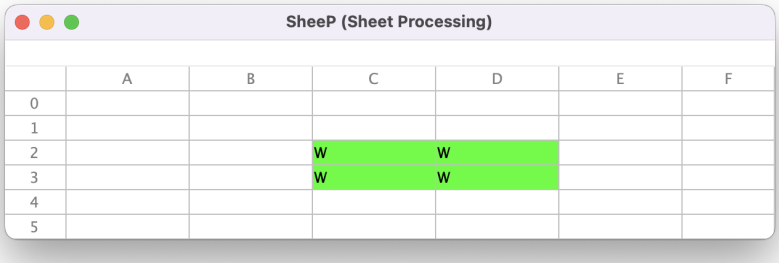
sheep.fun Provided classes that pre-populate spreadsheets with test data, such as the Fibonacci sequence using the `Fibonacci` class.

sheep.ui Provided implementation of the user interface.

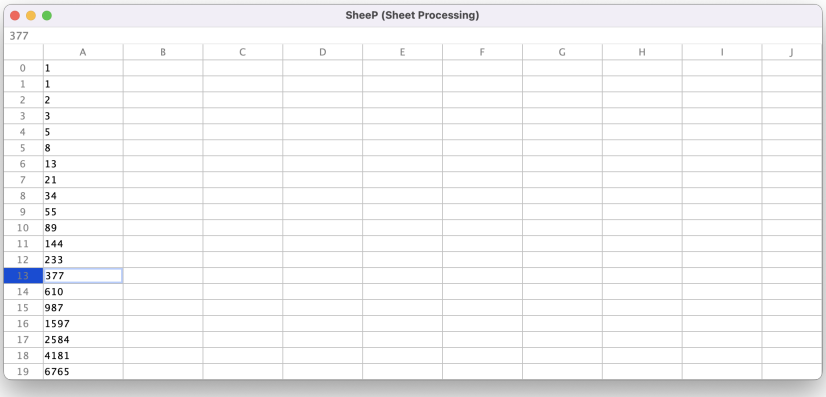
Stages

The assignment is split into stages to encourage incremental development. You should complete each stage before moving on to the next . The provided `Main` class allows you to run each stage individually by uncommenting the appropriate lines in the `main` method. Figure 1 highlights the classes that you will implement in each stage: green for stage 0, blue for stage 1, yellow for stage 2, and purple for provided code. At each stage, ensure that you thoroughly test your implementation.

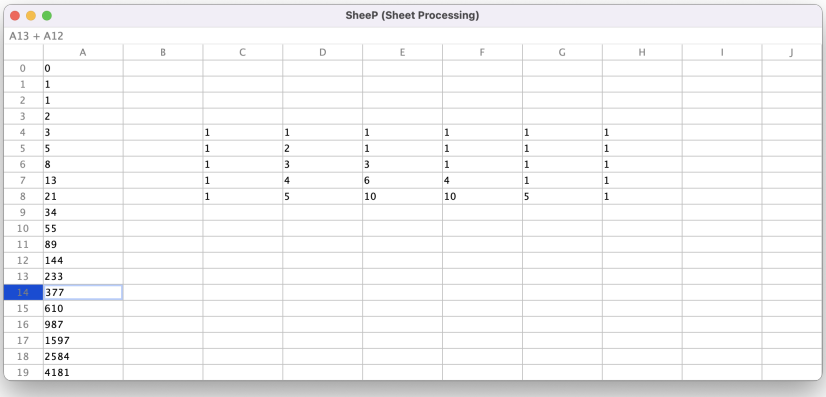
> **Stage 0** Create a simple implementation of a spreadsheet, `FixedSheet`. The `FixedSheet` class should be within the `sheep.sheets` package and implement the `sheep.core.SheetView` and `sheep.core.SheepUpdate` interfaces. After implementing the `FixedSheet` class and uncommenting the appropriate lines in the `main` method, the program should run as below.



Stage 1 Implement just the basic types of expressions within the spreadsheet: constant values, references to other cells, and empty cells. Create an expression factory to create these expressions and a parser to parse expressions from strings. Finally create `DisplaySheet` to show the results of these expressions. When the appropriate lines in the `main` method are commented out, the program should run as below.



Stage 2 Complete the implementation of expressions to include arithmetic operations. Your parser and expression factory should be able to parse and create these expressions. Create the full `Sheet` implementation, this sheet should appropriately update cells when other cells change. When the appropriate lines in the `main` method are commented out, the program should run as below.



Stage 3 If you have not already done so, ensure you write the required JUnit test classes.

Marking

The assignment will be marked out of 100. The marks will be divided into three categories: functionality (F), JUnit tests (T), and style (S).

	Weight	*Description*
F	65	The program is functional with respect to the specification.
T	25	JUnit tests can distinguish between correct and incorrect implementations.
S	10	Code style conforms to course style guides.

The overall assignment mark is defined as

$$A_1 = (65 \times F) + (10 \times S) + (25 \times T)$$

Functionality Each class has a number of unit tests associated with it. Your mark for functionality is based on the percentage of unit tests you pass. Assume that you are provided with 10 unit tests for a class, if you pass 8 of these tests, then you earn 80% of the marks for that class. Classes may be weighted differently depending on their complexity. Your mark for the functionality, F, is then the weighted average of the marks for each class,

$$F = \frac{\sum_{i=1}^{n} w_i \cdot \frac{p_i}{t_i}}{\sum_{i=1}^{n} w_i}$$

where n is the number of classes, w_i is the weight of class i, p_i is the number of tests that pass on class i, and t_i is the total number of tests for class i.

JUnit Tests The JUnit tests that you provide in `CellLocationTest` and `ReferenceTest` will be used to test both correct *and* incorrect (faulty) implementations of the `CellLocation` and `Reference` classes. Marks will be awarded for distinguishing between correct and incorrect implementations.[2] A test class which passes every implementation (or fails every implementation) will likely get a low mark.

Your tests will be run against a number of faulty implementations of the classes you are testing. Your mark for each faulty solution is binary based on whether at least one of your unit tests fails on the faulty solution, compared against the correct solution. For example, if you write 14 unit tests for a class, and 12 of those tests pass on the correct solution and 11 pass on a faulty solution, then your mark for that faulty solution is 1 (a pass). In general, if b_i of your unit tests pass on a correct implementation of class i and t_i pass on a faulty implementation, then your mark for that faulty solution is

$$f_i = \begin{cases} 1 & \text{if } b_i > t_i \\ 0 & \text{otherwise} \end{cases}$$

$$T = \frac{\sum_{i=1}^{n} f_i}{n}$$

where n is the number of faulty solutions.

See Appendix B for more details.

[2] And getting them the right way around.

Code Style The Code Style category is marked starting with a mark of 10. Every occurrence of a style violation in your solution, as detected by *Checkstyle* using the course-provided configuration[3], results in a 1 mark deduction, down to a minimum of 0. For example, if your code has 2 checkstyle violations, then your mark for code quality is 8. Note that multiple style violations of the same type will each result in a 1 mark deduction.

$$S = \frac{max(0, 10 - v)}{10}$$

where v is the number of style violations in your submission.

Note: There is a plug-in available for *IntelliJ* which will highlight style violations in your code. Instructions for installing this plug-in are available in the Java Programming Style Guide on Blackboard (Learning Resources → Guides). If you correctly use the plug-in and follow the style requirements, it should be relatively straightforward to get high marks for this section.

Electronic Marking

Marking will be carried out automatically in a Linux environment. The environment will not be running Windows, and neither IntelliJ nor Eclipse (or any other IDE) will be involved. OpenJDK 21 with the JUnit 4 library will be used to compile and execute your code and tests. When uploading your assignment to Gradescope, ensure that Gradescope says that your submission was compiled successfully.

Your code must compile.
If your submission does not compile, **you will receive zero marks**.

Submission

Submission is via Gradescope. Submit your code to Gradescope *early and often*. Gradescope will give you some feedback on your code, but it is not a substitute for testing your code yourself.

You must submit your code *before* the deadline. Code that is submitted after the deadline will **not** be marked (1 nanosecond late is still late). See Assessment Policy.

You may submit your assignment to Gradescope as many times as you wish before the due date. Your last submission made before the due date will be marked.

What to Submit Your submission should have the following internal structure:

`src/`	folders (packages) and `.java` files for classes described in the Javadoc.
`test/`	folders (packages) and `.java` files for the JUnit test classes.

A complete submission would look like:

[3]The latest version of the course *Checkstyle* configuration can be found at `http://csse2002.uqcloud.net/checkstyle.xml`. See the Style Guide for instructions.

```
src/sheep/expression/ExpressionFactory.java
src/sheep/expression/InvalidExpression.java
src/sheep/expression/Expression.java
src/sheep/expression/TypeError.java
src/sheep/expression/CoreFactory.java

src/sheep/expression/basic/Reference.java
src/sheep/expression/basic/Constant.java
src/sheep/expression/basic/Nothing.java

src/sheep/expression/arithmetic/Arithmetic.java
src/sheep/expression/arithmetic/Equal.java
src/sheep/expression/arithmetic/Divide.java
src/sheep/expression/arithmetic/Less.java
src/sheep/expression/arithmetic/Plus.java
src/sheep/expression/arithmetic/Minus.java
src/sheep/expression/arithmetic/Times.java

src/sheep/sheets/DisplaySheet.java
src/sheep/sheets/CellLocation.java
src/sheep/sheets/Sheet.java
src/sheep/sheets/FixedSheet.java
src/sheep/sheets/SheetBuilder.java

src/sheep/parsing/Parser.java
src/sheep/parsing/ParseException.java
src/sheep/parsing/SimpleParser.java

test/sheep/sheets/CellLocationTest.java
test/sheep/expression/basic/ReferenceTest.java

ai/README.txt
ai/*
```

Ensure that your classes and interfaces correctly declare the package they are within. For example, `Reference.java` should declare `package sheep.expression.basic;`.

Do not submit any other files (e.g. no `.class` files).
Note that `CellLocationTest` and `ReferenceTest` will be compiled individually against a sample solution without the rest of your test files.

Provided tests A small number of the unit tests (about 10-20%) used for assessing Functionality (F) are provided in Gradescope, which can be used to test your submission against. In addition, a small number of the JUnit faulty solutions used for assessing JUnit tests (T) are also provided in Gradescope.

The purpose of this is to provide you with an opportunity to receive feedback on whether the basic functionality of your classes and tests is correct or not. Passing all the provided unit tests does *not* guarantee that you will pass all the tests used for functionality marking.

Assessment Policy

Late Submission Any submission made after the grace period (of one hour) will not be marked. Your last submission before the deadline will be marked.

Do not wait until the last minute to submit the final version of your assignment. A submission that starts before 13:00 but finishes after 13:00 will not be marked.

Extensions If an unavoidable disruption occurs (e.g. illness, family crisis, etc.) you should consider applying for an extension. Please refer to the following page for further information:

`https://uq.mu/rl551`

All requests for extensions must be made via my.UQ. Do not email your course coordinator or the demonstrators to request an extension.

Remarking If an *administrative error* has been made in the marking of your assignment (e.g. marks were incorrectly added up), please contact the course coordinator (csse2002@uq.edu.au) to request this be fixed.

For all other cases, please refer to the following page for further information:

`https://uq.mu/rl552`

Change Log

Revision: 1.0.0

If it becomes necessary to correct or clarify the task sheet or Javadoc, a new version will be issued and an announcement will be made on the Blackboard course site. All changes will be listed in this section of the task sheet.

A Critical Mistakes

THINGS YOU MUST AVOID

This is being heavily emphasised here because these are critical mistakes which must be avoided.

Code may run fine locally on your own computer in IntelliJ, but it is required that it also builds and runs correctly when it is marked with the electronic marking tool in Gradescope. Your solution needs to conform to the specification for this to occur.

- Files must be in the correct directories *(exactly)* as specified by the Javadoc. If files are in incorrect directories *(even slightly wrong)*, you may lose marks for functionality in these files because the implementation does not conform to the specification.
- Files must have the exact correct package declaration at the top of the file. If files have incorrect package declarations *(even slightly wrong)*, you may lose marks for functionality in these files because the implementation does not conform to the specification.
- You must implement the public and protected members exactly as described in the supplied documentation (*no extra public/protected members or classes*). Creating public or protected data members in a class when it is not specified will result in loss of marks, because the implementation does not conform to the specification.
    - You are encouraged to create private members as you see fit to implement the required functionality or improve the design of your solution.
- Never import the `org.junit.jupiter.api` package. This is from JUnit 5. This will cause the JUnit tests to fail, resulting in no marks for the JUnit component.
- Do not use any version of Java newer than 21 when writing your solution. If you accidentally use Java features which are only present in a version newer than 21, then your submission may fail to compile.

B JUnit Test Marking

The JUnit tests you write for a class (e.g. `CellLocationTest.java`) are evaluated by checking whether they can distinguish between a:

correct implementation of the respective class
e.g. `CellLocation.java`, made by the teaching staff, and

incorrect implementations of the respective class
"deliberately made (sabotaged) by the teaching staff".

First, we run your unit tests (e.g. `CellLocation`Test.java) against the correct implementation of the respective classes (e.g. `CellLocation.java`).

We look at how many unit tests you have, and how many have passed. Let us imagine that you have 7 unit tests in `CellLocationTest.java` and 4 unit tests in `ReferenceTest.java`, and they all pass (i.e. none result in `Assert.fail()` in JUnit4).

We will then run your unit tests in both classes (`CellLocationTest.java`, `ReferenceTest.java`) against an incorrect implementation of the respective class (e.g. `CellLocation.java`). For example, the `foo()` method in the `CellLocation.java` file is incorrect.

We then look at how many of your unit tests pass.

`ReferenceTest.java` should still pass 4 unit tests.
However, we would expect that `CellLocationTest.java` would pass **fewer than** 7 unit tests.

If this is the case, we know that your unit tests can identify that there is a problem with this specific (incorrect) implementation of `CellLocation.java`.

This would get you one identified faulty implementation towards your JUnit mark.

The total marks you receive for JUnit is the number of identified faulty implementations, out of the total number of faulty implementations which the teaching staff create.

For example, if the teaching staff create 10 faulty implementations, and your unit tests identify 6 of them, you would receive 6 out of the 10 possible marks for JUnit, or 15 marks when scaled to 25%.

There are some limitations on your tests:

1. If your tests take more than 20 seconds to run, or
2. If your tests consume more memory than is reasonable or are otherwise malicious,

then your tests will be stopped and a mark of zero given. These limits are very generous (e.g. your tests should not take anywhere near 20 seconds to run).

C Generative Artificial Intelligence

While the use of generative AI for this assignment is discouraged, if you do wish to use it, ensure that it is declared properly.

For this, you must create a new folder, called "`ai`", within this folder, create a file called "`README.txt`". This file must explain and document how you have used AI tools. For example, if you have used ChatGPT, you must state this and provide a log of questions asked and answered using the tool.

The "`README.txt`" file must provide details on where the log of questions and answers are within your "`ai`" folder.

If you plan to use continuous AI tools such as Copilot, you must ensure that the tool is logging it's suggestions so that the log can be uploaded. For example, in IntelliJ, you should enable the log by following this guide: `https://docs.github.com/en/copilot/troubleshooting-github-copilot/viewing-logs-for-github-copilot-in-your-environment` and submit the resulting log file.