- Our submission has one main folder containing this report, a gui/ folder, a github/ folder, and a coverage/ folder. The gui/ folder contains the GUI tests, as described below, and the github/ folder contains the whitebox/blackbox tests, as described below. The coverage/ folder contains the HTML Jacoco coverage report

Requested Deliverables:

1. Test Artifacts: see testFiles/ folder in the Stirling-PDF project. It has inputs/ and outputs/ folders for the required files to test, as well as the generated documents for inspection.

2. The SUT:
   a. Included in our submission under github/
   b. Available here: https://github.com/Stirling-Tools/Stirling-PDF
   c. The user-facing SUT is available here (used for GUI testing): https://hub.docker.com/r/frooodle/s-pdf

3. Test files
   a. Included in our submission folder (GUI tests under gui/ ; Whitebox/blackbox under github/)
   b. Instructions to run are below in this document

4. Presentation slides: https://docs.google.com/presentation/d/1F5bMYMIseOJkTZVN5XY3qSBwOsK-kkCMAQuQ-zi4dTY/edit?usp=sharing

5. Steps to run tests: see section below

Instructions to Run

GUI

- Pull the Docker image for StirlingPDF

    - Run “`docker pull frooodle/s-pdf`”
    - https://hub.docker.com/r/frooodle/s-pdf

- Spin up the Docker image
    - Run “`docker run -d -p 8080:8080 -e DOCKER_ENABLE_SECURITY=false --name stirling-pdf frooodle/s-pdf:latest`”
    - See the docker hub link above for details if needed

- Open the GUI testing selenium IntelliJ project (under the “GUI” folder in our submission) using IntelliJ [IntelliJ -> open project -> select the “PDFSeleniumTests” folder under the GUI folder in our submission]

- The main test file is under [src -> test -> java -> PdfGuiTests.java]
    - These tests can be run by clicking the “run” button in the IntelliJ gutter, as we have done in class

Backend Tests

- Open the project in the ____ folder
- Ensure Java 17 and a Java 17 JVM are installed
- Right click the src/test folder and run all the tests
- To generate a coverage report, run: gradlew jacocoTestReport
    - Navigate to the index.html in build/reports/jacoco to view the report

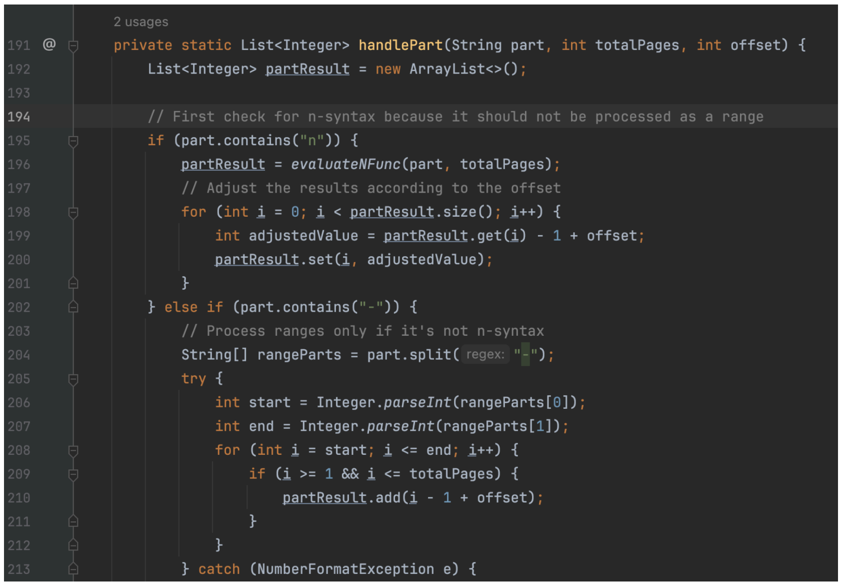
Note: there should be 265 tests that run, and 227 that pass (there were 15 existing tests, so we added 250)

Faults

The blackbox and whitebox tests that fail have been marked with @Tag(“fails”) and a comment that explains the failing case.

Still, we uncovered the following faults:

Backend

- There was no documentation in the PDF utility functions at all
- Converting a multi-page PDF to a single .tif image does not work (produce blank pdf)
- Inconsistent 0/1 based indexing for page parsing. See the following code snippet:
    - 

```
    2 usages
191 private static List<Integer> handlePart(String part, int totalPages, int offset) {
192     List<Integer> partResult = new ArrayList<>();
193
194     // First check for n-syntax because it should not be processed as a range
195     if (part.contains("n")) {
196         partResult = evaluateNFunc(part, totalPages);
197         // Adjust the results according to the offset
198         for (int i = 0; i < partResult.size(); i++) {
199             int adjustedValue = partResult.get(i) - 1 + offset;
200             partResult.set(i, adjustedValue);
201         }
202     } else if (part.contains("-")) {
203         // Process ranges only if it's not n-syntax
204         String[] rangeParts = part.split( regex: "-");
205         try {
206             int start = Integer.parseInt(rangeParts[0]);
207             int end = Integer.parseInt(rangeParts[1]);
208             for (int i = start; i <= end; i++) {
209                 if (i >= 1 && i <= totalPages) {
210                     partResult.add(i - 1 + offset);
211                 }
212             }
213         } catch (NumberFormatException e) {
```

    - Note that offset is meant to account for 0/1 based indexing, but when the “String part” is 0, it is completely ignored due to line 209 only considering indices >= 1
- In PdfUtils, containsTextInFile, pageCount, and pageSize close the input document, while hasImages and hasText do not. This is inconsistent and confusing.
- PdfUtils.hasImages calls the PDFBox function “getPage(i)” in a loop, which is very inefficient. The PDFBox documentation explicitly states that “getPages()” should be used in this case, because getPage() implicitly works like “getPages().get(i)” for each call.
- Converting to a PDF (imageToPdf()), and changing the color type (to blackwhite or greyscale) works fine, but the metadata is not preserved. Checking the type of the image returns the code for rgb, rather than blackwhite or greyscale.
- The JPEGFactory use (see code in presentation) seems to be lossy in converting JPEGs to PDFs.
- PdfUtils.AddImageToDocument has an if … else if block for handling the fitOption. These blocks handle “fillPage”, “fitDocumentToImage”, and “maintainAspectRatio”. If the fitOption is none of these three, there is no “else” block to handle it. An error is not thrown, and the function simply returns clearly. There is no documentation, but we consider this an oversight, a fault, and therefore undesired behavior.

- GeneralUtils.isValidUrl is not robust enough and ends up validating very clearly malformed url’s (ie url’s with a space in them, with two consecutive dots in them, with invalid domain, with invalid protocol)
- A number of methods throw NullPointerExceptions. Some get thrown when the methods are passed PDF’s that contain blank pages. Moreover, many methods specify a certain type of exception to be thrown (such as IOException) but throw other types under many circumstances. Some of these methods are: PdfUtils.hasImages(), PdfUtils.hasImagesOnPage(). A number of other methods just fail to assert that the passed parameters are not null before working with them.
- Ideally when converting an image to a pdf and then back to an image, the starting and ending images should be equal. We tested this Property using PdfUtils.imageToPdf() and PdfUtils.convertFromPdf() and saw that the test failed, meaning that these conversions are leading to some pixel loss.
- We found some classes which were completely empty, some methods which were unused and also some methods which were redundant

GUI

- There were 3 distinct faults found with GUI testing. They are tagged in the main gui testing file [src -> test -> java -> PdfGuiTests.java] with the tag “Fault”. These tests have comments above them describing what the fault was. For clarity, those comments are repeated here:

- Fault 1 - landing page: when a tool on the main page is "favorited", it moves to the top of the list/container. When it is unfavorited, it remains at the top of the list/container. However, when the page is refreshed, the tool's card goes back to its original position in the list. So, the intended behavior was for the card to go back to its original position immediately upon being unfavorited, without needing a refresh. It does not do this.

- Fault 2 - PDF to Image tool page: When a pdf is submitted to be converted to an image, an error message is shown. This happens regardless of the type. This is an issue with the **integration** of the PdfToImage functionality with the UI. This is a fault because the intended behavior is for the pdf to be converted to an image, not for an error message to be shown.

- Fault 3 - PDF to Image tool page: When the error message (described in fault 2) is closed and the submission is re-attempted, the page just stalls and the button says "processing...." forever. This is a fault because the intended functionality is for the error message to be shown again. The fact that it stalls at “processing...” misleads the user into thinking the process is working.

Notes

We chose to test the code that contained the main functionalities related to the PDF tools, which is why we tested classes in the utils folder as opposed to the API routes or the models.
One of the main challenges with this project was that we were dealing with the manipulation of different file formats, such as PDF, JPEG, PNG and TIF. For our tests, we worked with locally saved files for each of these formats, as well as Objects from the PDFBox library and file mocks we created using Mockito. We tested the equivalence of these objects manually, through assert statements and also using Property testing.

PDF documents were not only challenging to manipulate, but to verify as well. We found ourselves having to answer questions such as “how can we assert that the converted PDF (as an image) represented the original document?”. There are many things that can go wrong, and only a select few can be tested for. The file may not be corrupted, but it may be missing some or all elements from the original. The same goes for conversions in the other direction. Pixel loss is normal, as are differences in representation. Does it matter if the image has a different underlying type of rgb/blackwhite/greyscale if it looks the same? All these open ended criteria made our task more challenging.

Coverage

(html report also included in submission)

The classes we tested were stirling.software.SPDF.utils.
- PdfUtils
- GeneralUtils
- ErrorUtils

- ImageProcessingUtils
- RequestUriUtils

We quickly realized that 100% statement and branch coverage for the chosen Utils classes would not be attainable. So we pivoted to aim for > 90% for both metrics in the tested class (we surpassed this).
The only statements that are not covered are those that are unreachable, or very hard to reach. For example, a catch block that can be triggered by file operations or file system operations failing. These failures are very hard to trigger, and may result from a filesystem not having enough storage, memory, etc. to handle file creation or saving. These were very hard to trigger.

Branches that were not covered were almost exclusively unreachable branches, due to short circuiting && conditions, or mutually exclusive conditions. For example, in line 410 of PdfUtils, “i==0” will never be false when the first condition “!everyPage” is true, due to the break statement that exits the loop after the first page.

GUI

- Instructions to run are at the top of this document - the rest of the notes assume those have been followed

- The main test file [src -> test -> java -> PdfGuiTests.java] has many comments at the top of the file that explain everything that is needed. Some of those are repeated here, but for the full context, please see that file

- The class tests the main landing page [http://localhost:8080/ ; the docker instructions above spin up the localhost endpoint] and the “PDF To Image” sub-tool [http://localhost:8080/pdf-to-img]

- The landing page is tested thoroughly with respect to 3 sub-modules (View PDF, Remove, and Auto Redact) to make the main page tests substantial and generalizable

- Note that there is no coverage criterion "number" to hit here - the goal of this is to ensure that things that must work on the site 1) do work as of now, and 2) do not break when future changes are added (i.e. these tests, especially with the page object model, should not fail when new features are added)

- The tests that fail are ones that reveal faults. These tests are tagged with the tag "Fault". See the “Faults” section of this document

- There are 2 page object model files - see the main test file comments for details

- In general, the name of each test describes what it does