CITS5501 Project 2022

Version:	0.3
Date:	12 May, 2022

Changes

0.3 – 12 May, 2022

- Removed stray backslashes from string in the "Parser class" section ("a line\\\n")
- Added section "Documenting assumptions" in "Items to submit"

0.2 – 9 May, 2022

- Removed '~~A "line feed" character (Unicode `U+000A`)~~' from "Allowable input" to resolve an inconsistency in the spec. Thanks to Alex Brown for spotting this.

Introduction

- This project contributes **35%** towards your final mark this semester, and is to be completed as individual work.
- The project is marked out of 50.

- The deadline for this assignment is **5:00 pm Thu 26 May**
- You are expected to have read and understood the University Guidelines on Academic Conduct. In accordance with this policy, you may discuss with other students the general principles required to understand this project, but the work you submit must be the result of your own effort.
- You must submit your project before the submission deadline above. There are significant Penalties for Late Submission (click the link for details).

Items to submit

You will need to submit a `.zip` file of Java code, and a PDF report containing answers to questions (marks will be deducted for documents submitted in other formats).

Your PDF report should use A4 size pages. The font for body text should be between 9 and 12 points. The report should contain numbered headings, with useful heading titles. Each question should be answered on a new page. Any diagrams, charts or tables used must be legible and large enough to read when viewed on-screen at 100% magnification.

All pages (except the cover, if you have one) should be numbered. If you give scholarly references, you may use any standard citation style you wish, as long as it is consistent. Cover sheets, diagrams, charts, tables, bibliographies and reference lists do not count towards any page-count maximums.

Documenting assumptions If you believe there is insufficient information for you to complete part of this project, you should document any *reasonable assumptions* you had to make in order to answer a question or write a method.

In that case, make sure your report has as one of its first sections a section entitled "Assumptions", in which you list these, giving each one a number and explaining why you think it's a reasonable assumption.

Then in your code or later in your report, you can briefly refer to these assumptions (e.g. "This test case assumes that Assumption 1 holds, so that we can . . . ").

Background

Sometimes when creating a document, it is useful to be able to execute code or commands (for instance, Python code, or shell script, or MS Windows batch code) which generate part of the document. An example of a "file generation" or "document generation" program like this is Cog, written in Python.

For this project, you will be working on a document generator program called JDocGen, written in Java. The program takes as input a text file that includes specially marked sections ("code chunks") containing shell commands to be run, runs the commands contained in those sections, and splices their output into a new version of the document.

A code chunk is started by a line in a text file containing just the character sequence `"=<<<"`, and ends either (a) after a line containing just the character sequence `">>>"` or (b) if no such line is found, at the end of the text file.

Example

By way of example, suppose you were writing a tutorial on how to use file and directory commands from the command prompt. (In this example, a Linux command is shown being run, but similar commands could be run on any operating system.)

Part-way through the tutorial, you might want to show what the actual result of doing a directory listing is. You could do so by creating a text document (named `tutorial.txt`, say) like the following:

```
*File and directory tutorial*

by John A. Hacker

On Unix-like systems, typing the command "ls" will list the
contents of the current directory.
```

```
7
8  It will produce output that looks like this:
9
10 =<<<
11 ls
12 >>>
```

You could then supply your `tutorial.txt` file as input to JDocGen, which would run the `ls` command, insert the result of running that command into the relevant part of the document, and output the new version of the document.

If the current directory contained files `tutorial.txt` and `TODO.txt`, then the text produced by JDocGen would look like this:

```
1  *File and directory tutorial*
2
3  by John A. Hacker
4
5  On Unix-like systems, typing the command "ls" will list the
6  contents of the current directory.
7
8  It will produce output that looks like this:
9
10 TODO.txt tutorial.txt
```

Supplied files

You are supplied with code for the JDocGen program in a zip file, `jdoc.zip`. This code should compile with any current recent Java compiler or IDE (such as BlueJ, Eclipse or IntelliJ IDEA).

The `jdoc.zip` file contains a directory named `src`, where you will find the code for the JDocGen program, as well as a directory `test` containing some initial test classes.

You **should not** need to alter the code in the `src` directory, nor should you submit it (if you do so, it will simply be ignored). Any code you write should instead be included in the `test` directory, which you will need to submit as a `.zip` file.

The JavaDoc for the JDocGen classes can be viewed at https://cits5501.github.io/assignments/project/doc.

Specifications

In this section, we specify (in more detail than the JavaDoc comments do) how several of the JDocGen classes should behave.

We call a line containing just the character sequence `"=<<<"` a "start marker line", and a line containing just the character sequence `">>>"` an "end marker line".

Allowable input

A "line" in the input document is considered to end when either

a. a newline ('\n') character is reached, or
b. the end of the document is reached.

That is, the final line of the input document may or may not end in a newline character.

The input document should not contain any Unicode line end characters besides newline characters, or the result of processing the document is undefined. In particular, the input document may not contain any of the following characters:

- A "vertical tab" character (Unicode `U+000B`)
- A "form feed" character (Unicode `U+000C`)
- A "carriage return" character (Unicode `U+000D`)
- A "next line" character (Unicode `U+0085`)
- A "line separator" character (Unicode `U+2028`)
- A "paragraph separator" character (Unicode `U+2029`)

Parser class

The Parser class splits a document up into chunks; each chunk is either a "doc chunk" or a "code chunk".

1. The result of parsing an empty document should be a `List<Chunk>` containing 0 items.

2. When a code chunk is parsed, its contents should be all lines *between* the start marker line and the end marker line; the marker lines should not be included in the contents.

For example, parsing the following document:

```
a line
=<<<
echo XXX
>>>
last line
```

should result in a `List<Chunk>` containing three items:

- A doc chunk with the contents `"a line\n"`
- A code chunk with the contents `"echo XXX\n"`
- A doc chunk with the contents `"last line\n"`

Processor classes

A processor class can be seen as taking as input a `List<Chunk>` containing 0 or more chunks; possibly transforming the contents of each chunk in some way (for instance, by running the commands found in a "code chunk", and replacing the chunk content with the output of those commands); and then joining the content of the transformed chunks together to produce an output document.

The first chunk in the input `List<Chunk>` may be either a doc chunk or a code chunk. The input may not contain two consecutive chunks of the same type; that is, it should contain alternating doc chunks and code chunks. The result of processing input which does contain consecutive chunks of the same type is undefined.

The last line of an output document emitted by a processor class should always end in a newline ('\n'); if the contents of the last transformed chunk does *not* end in a newline, one is added.

Tasks

Some of the following tasks require you to write answers to questions; others require you to write code. Any code you write should go into the `test` directory of the supplied `JDocGen` and be submitted as a `.zip` file.

Answers you write should be submitted as a PDF file. (Note that marks will be deducted for files submitted in other formats.)

A. Parser – ISP test design

One of your colleagues, Anastasia, is writing a test plan for the `Parser` class using input space partitioning. She suggests that the `Parser` class can be viewed as a function which takes as input a string (i.e., a document), and outputs a `List<Chunk>`. She proposes to use the interface-based approach to input space partitioning, with the aim of achieving Base Choice Coverage. She suggests the following, single characteristic for the input string:

- Is the string empty? (This divides strings into two partitions, empty vs non-empty.)

For empty strings, there is only one possible value. For non-empty strings, Anastasia suggests a simple, two-line string (`"This is a sample document.\nIt contains two lines\n"`) as the base choice, as well as several non-base choices:

- A very short document (just the string `"This"`)
- A document containing non-English Unicode characters
- A document containing a hundred lines (consisting of the base choice string, repeated 50 times)

Answer the following question in your report:

- Is your colleague's approach to applying the ISP technique an appropriate one? Why or why not? How would you apply the technique? You should give each of the steps involved in applying ISP, except that for the last step (refining into test values), you need give only 3 test cases. (10 marks)
- Your 3 test cases may *not* include the case where the input string is the empty document.

B. Parser – ISP test code

The `parsers.ParserTest` contains partial code for testing the `Parser` class. Implement your test cases from section A by adding them to the `ParserTest` class. (10 marks)

(You can remove the `testDoc` test or leave it in, as you prefer – it will be ignored by markers.)

C. Parser – syntax-based testing

Your supervisor, Yennefer, suggests that because input documents use a special syntax (start marker lines and end marker lines) to demarcate code chunks, syntax-based testing could be used to come up with tests, and assess the coverage level of existing tests.

In your report,

- Give a grammar which describes the syntax of input documents. Explain how it works and how you derived it. (7 marks)

Hint: You may assume you are allowed to use a special non-terminal symbol, "`<nonmarker_text>`". It expands into a string of arbitrary length which never contains the character sequence `"=<<<"` nor the character sequence `">>>"`.

D. ShellProcessor tests

The `processors.ShellProcessorTest` contains a sample test, `testOneLineDoc` for the `ShellProcessor` class.

In your report, answer the following question:

- Do you think `testOneLineDoc` is a unit test, or an integration test? Why? (3 marks)

In the `processors.ShellProcessorTest` class:

- Create a JUnit *parameterized* test in the `ShellProcessorTest` class, which executes on 4 different sets of test values. You need not follow any particular technique for choosing your test values, but we will assess how likely it is that your test values will identify a wide range of defects. (10 marks)

In your report, answer the following question:

- For your 4 different sets of test values, why did you choose them? What sort of defect do they aim to expose? (5 marks)

E. Logic-based tests

Your colleague Anastasia has been reading about logic-based testing, and wonders whether it would be a good use of time to assess what level of coverage the tests for the JDocGen system have (e.g. Do they have Active Clause Coverage)?

In your report, answer the following question:

- Would measuring logic-based coverage for the JDocGen be a good use of the team's time? Why or why not? Explain your reasoning. (5 marks)