CITS5501 Project 2023

Version:	0.2
Date:	12 May, 2022

Changes in version 0.2

- Java code: `GladiusException` base class changed from `Exception` to `RuntimeException`.
- Long answer citation style suggested.
- Good coding practices expanded on.
- Syntactic vs semantic checks clarified.
- Question 1: start symbol specified.
- Question 4: correct name of method.
- Question 6: requirements clarified.
- Question 9: removed; total mark changed from 60 to 50.

1 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 25 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).

2 Clarifications and changes to the project specification

You are encouraged to start reading through this project specification and planning your work as soon as it is released. Any queries regarding the project should be posted to the Help5501 forum with the "project" tag.

Any clarifications or amendments that need to be made will be posted by teaching staff in the Help5501 forum.

3 Format and submission of your work

All code and answers to questions should be submitted by making a Moodle submission.

For any long English answers:

- Please structure your answers using numbered headings where appropriate. You may use Markdown to format your answers. Markdown will be rendered into HTML for marking using the "Python-Markdown" package.
- If you include any diagrams, charts or tables, they must be clear, legible and large enough to read when viewed on-screen at 100% magnification.
- If you give scholarly references, you may use any standard citation style you wish, as long as it is consistent. However, a recommended citation style is "AMS short alpha-numeric" (see AMS style guide, sec 10.3).
- Diagrams, charts, tables, bibliographies and reference lists do not count towards any word-count maximums.

Submitted code should meet the usual guidelines for CITS5501 code:

- code should be clearly written, well-formatted, and easy for others to understand
- function bodies should not contain excessive inline comments
- code should compile without errors or warnings
- code should follow sound programming practices, including:
  - the use of meaningful comments
  - well chosen identifier names
  - appropriate choice of basic data-structures, data-types, and functions
  - appropriate choice of control-flow constructs
  - proper error-checking of any library functions called, and
  - cleaning up/closing any files or resources used.

Any methods (including JUnit `@Test` methods) that you write should include a Javadoc documentation block which follows the guidelines at https://www.oracle.com/au/technical-resources/articles/java/javadoc-tool.html under "Writing Doc Comments".

Note that code which does not compile will be awarded (at most) a very small number of marks. You can check that your code compiles using Moodle; teaching staff will not fix your code if it does not compile.

Your code should never emit any output to `System.out` or `System.err` unless specified in the question. Doing so is extremely poor practice, and will typically result in your code being awarded a low number of marks (and in any case, will likely result in your code failing any checks or tests applied by Moodle).

3.1 Documenting assumptions

If, after posting questions to the Help5501 forum and having received answers, you believe there is still 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 portion of code.

Moodle will include a question entitled "Assumptions" in which you can list these. Make sure you give each assumption a number, and explain why you think it's reasonable.

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

An assumption which contradicts anything in the project specification, the Java class specifications or postings made by staff in the Help5501 forum is *prima facie* not reasonable.

4 Background

Your software development team at Falchion Enterprises is developing a new interface for advanced users of Falchion's travel reservation system, GLADIUS. The GLADIUS reservation system already exists and normally is accessed through a web interface. However, research has shown that advanced users can be much more productive using a terminal-based interface, which your team is currently prototyping.

The following section describes the two commands that are being prototyped, `shop flight fare` and `air book req`.

4.1 GLADIUS command description

GLADIUS commands make use of a number of formats specified below. Some of these formats have rules for both syntactic validity (whether they have the correct "form") and semantic validity. To make syntax testing simpler and more tractable, simplified syntax testing rules are suggested in several cases.

Airport codes Airport codes are 3-letter IATA codes for airports – such as SYD (Sydney), PER (Perth), MEL (Melbourne) and CBR (Canberra). For *syntax testing purposes only*, you do not need to validate codes against the exact full list of these 3-letter codes; you may make the simplifying assumption that any 3-letter sequence of the letters from 'A' through to 'D' inclusive specify an IATA code.

For any other purposes, an airport code must be 3 capital letters in the range 'A' to 'Z' inclusive in order to pass syntax validity checks, and must be looked up in a list of current IATA airport codes to be semantically valid. You can use the list provided in `iata_codes.zip` at https://cits5501.github.io/assessment/#project to come up with semantically valid or invalid airport codes. (In actuality, an "airport code validator" would likely be a class that could be mocked, but we won't make use of such a class.)

Cabin types Cabin types are 1-letter codes specifying a seating location in a plane: P (premium first class), F (first class), J (premium business class), C (business class), S (premium economy class), and Y (economy class).

Currency codes Currency codes are 3-letter codes specifying a currency as per ISO standard 4217. For *syntax testing purposes only*, you may make the simplifying assumption that any 3-letter sequence of the letters from 'E' through to 'H' inclusive specifies an ISO 4217 currency code. (For example: 'EFG' would be a valid currency code, for syntax testing purposes.)

For other purposes, you may assume that a currency code must be 3 capital letters in the range 'A' to 'Z' inclusive in order to pass syntax validity checks, and must be looked up in a list of current ISO 4217 currency codes to be semantically valid. You can use the list provided on the website at https://cits5501.github.io/assessment/#project to come up with semantically valid or invalid currency codes. (In actuality, a "currency code validator" would likely be a class that could be mocked, but we won't make use of such a class.)

Dates A date is always specified in the form `YYYY-MM-DD`. To be syntactically valid, each character in a date (other than the hyphens) must be a digit in the range '0' to '9' inclusive. (For example: `0001-00-00` counts as a syntactically valid date.) To be semantically valid, the date must be a valid date in the Gregorian calendar.

Datetimes A datetime is a date, followed by the letter "T", and a time in the format `HH:MM:SS`. To be syntactically valid, each character in a time (other than the colons) must be a digit in the range '0' to '9' inclusive. (For example: `99:99:99` counts as a syntactically valid time.) To be semantically valid, the time must be a valid 24-hour clock time.

Airline codes An airline code is a 2-character IATA airline code. For *syntax testing purposes only*, you may make the simplifying assumption that any 2-letter sequence of the letters 'I' through to 'L' inclusive specifies an IATA airline code.

For any other purposes, an airline code must be 2 characters in the range 'A' to 'Z', 'a' to 'z' or '0' to '9' inclusive in order to pass syntax validity checks, and must be looked up in a list of current IATA airport codes to be semantically valid. You can use the list provided in `iata_codes.zip` at https://cits5501.github.io/assessment/#project to come up with semantically valid or invalid airline codes. (In actuality, an "airline code validator" would likely be a class that would be mocked, but we won't make use of such a class..)

Flight numbers A flight number is a 2-character airline code, followed by 1 to 4 digits. (For example: assuming 'II' is a valid airline code, then 'II1234' would count as a syntactically valid flight number.)

"For syntax testing purposes only" means "For the purposes of questions 1–4 and 8–9".

The commands you need to consider are listed below. Each command has a name, consisting of multiple words separated by whitespace. The command may have parameters; if it does, then after the command name should appear whitespace characters and then various parameters, described below. The command may have subcommands; if it does, then after a newline is entered, multiple subcommands can be given (one per line). The command finishes when the word `EOC` ("end of command") is entered on its own on a line.

Monospace font is used in the command descriptions to represent literal text. Parameters are represented in bold monospaced font (e.g. **`ORIGIN`**). Optional parameters are surround by square brackets (`[]`).

`shop flight fare`

This command shows the cheapest available flights between two airports, one flight per day, for a customizable number of days past the current date. For each date, flight details for the lowest-fare flight on that date are displayed. If multiple flights have the same fare, the earliest is displayed. If multiple flights have the same fare and the same departure datetime, the one with the lowest flight number (sorting lexicographically) is displayed.

The syntax for the `shop flight fare` command is as follows:

> `shop flight fares` **`ORIGIN DESTINATION TRIP_TYPE`** [**`LENGTH_OF_STAY`**] **`CABIN_TYPE DEPARTURE_DATE`**

Both the **`ORIGIN`** and **`DESTINATION`** must be 3-letter IATA codes. **`TRIP_TYPE`** should be one of the strings "OneWay" or "Return". When the **`TRIP_TYPE`** is "Return", a **`LENGTH_OF_STAY`** in days must be specified, which is a number from 0 to 20 inclusive. **`CABIN_TYPE`** must be a cabin type code. **`DEPARTURE_DATE`** must be a valid date of the form `YYYY-MM-DD`, specifying a date after, but no more than 100 days past the current date.

The command returns a numbered list of records. Each record displays the following information (in various customizable formats):

- CurrencyCode: an ISO 4217 code specifying the currency the price is in.
- DepartureDateTime: the datetime when the flight leaves.
- ReturnDateTime: if this is a return flight, the datetime when the return flight leaves.
- Fare: a number specifying the fair amount.
- Flight number: the flight number.

`air book req`

The "air book request" command reserves multiple "segments" of a flight, where each segment is a direct flight between two airports. (These must later be paid for, or the airline responsible will release the reservation. The time limit before reservations are released depends on the airline.)

The syntax for the `air book request` command consists of just the words `air book req`.

This is followed by a newline, and after the newline, multiple *segment* subcommands are specified, one per line; when no more segments are to be entered, the word "EOC" (end of command) should be entered on a line of its own.

Each segment subcommand has the following syntax:

> `seg` **`ORIGIN DESTINATION FLIGHT_NUMBER DEPARTURE_DATE CABIN_TYPE NUM_PEOPLE`**

Both the **`ORIGIN`** and **`DESTINATION`** must be 3-letter IATA codes. **`DEPARTURE_DATE`** must be a valid date of the form `YYYY-MM-DD`, and must be after the current date. **`CABIN_TYPE`** must be a cabin type code. **`NUM_PEOPLE`** is the number of seats to book, from 1 to 10 inclusive.

After a full `air book req` command (including any segment subcommands) is entered, the system should display either the message "`OK`", a space, then a 6-character confirmation code, or an error message.

4.2 GLADIUS command error messages

If a user issues an invalid command, the words "`ERROR`", a 3-digit code, and an English-language explanation of the problem should be printed.

A command could be invalid due to syntactic errors (e.g. a 4-letter sequence is given where a 3-letter airport code should have been given), or semantic ones (e.g. for the `shop flight fares` command, a date is specified which is more than 100 days in the future).

The error code is 100 for syntactic errors; 200 for semantic errors; and 300 if there were errors connecting to the GLADIUS servers. Other error codes are currently not used.

4.3 Supplied files

You are supplied with the Java code which your team has currently written for the new user interface – see the `gladius-source.zip` file which can be downloaded from Moodle.

This code should compile with any current recent Java compiler or IDE (such as BlueJ, Eclipse or IntelliJ IDEA).

A brief description of some of the classes in the Java code is given below:

Enumerated types: The code includes `CabinType` and `TripType` enum types.

Exceptions: The code includes `SyntacticError` and `SemanticError` exceptions, which are thrown by methods in the `CommandParser` interface and by constructors of `Command` sub-types.

Commands, processors and command responses: The code includes `AirBookRequestCommand` and `ShopFlightFareCommand` classes, which represent the two commands your team is prototyping.

These commands have `Command` as a base class. They can be passed to methods of the `CommandProcessor` interface, which typically represents a remote GLADIUS server.

Assuming the command can be processed, `CommandProcessor` returns a sub-type of the `CommandResponse` class – either `AirBookRequestResponse`, or `ShopFlightFareResponse`.

Detailed specifications for each class and method are provided as JavaDoc comments in the supplied code.

You should **not submit** any of the code in the `src` directory. The Moodle test servers will have their own versions of classes in that directory.

5 Tasks

Answes to the following questions should be submitted via Moodle. A more detailed marking rubric will also be available on Moodle. Some of the questions require you to write English answers; others require you to write code.

Question 1 [**4 marks**]

Give a BNF or EBNF grammar that will parse (or generate) valid GLADIUS commands. Your grammar should use the notation accepted by the BNF playground. It must be in plain text which could be pasted into the BNF playground and compiled without error.

The grammar should include a non-terminal `<gladius_command>` to be used as the start symbol for the grammar. The grammar should specify any whitespace that needs to appear between other symbols in the command. (As a simplifying assumption, you may assume that a single space character is sufficient as a separator between words.)

Question 2 [4 marks]

Write a static Java method, `countTerminalSymbols`, which counts the terminal symbols in a grammar.

The signature for the method should be: `static int countTerminalSymbols(List<String>)`.

The input will be a grammar in BNF or EBNF format (specifically, the notation accepted by the BNF playground), split into *meta-symbols* – each element of the `List<String>` will be one meta-symbol. Meta-symbols consist of terminal symbols (the double quote marks are included in the meta-symbol), non-terminal symbols (the angle brackets are included in the meta-symbol), newlines, and other meta-syntactic symbols used by the EBNF playground (such as the “is-defined-as” symbol, `::=` or the “alternative” or “or” symbol, `|`.)

For example, the following `List<String>` would represent a very small grammar consisting of just two strings, `sat` and `sun`:

```
List<String> myGrammar =
      List.of("<weekendDay>", "::=", "\"sat\"", "|", "\"sun\"");
```

The `countTerminalSymbols()` method would return the int `2` given the grammar `myGrammar`.

You may assume that `java.util.*` and `java.util.regex.*` import statements appear at the top of the source file for your code.

Question 3 [4 marks]

Would it be possible to exhaustively test the syntax of GLADIUS commands (that is, to write tests which have derivation coverage)? Why or why not? What about if we restricted ourselves to just the `shop flight fare` command? Give reasons for your conclusions. (Maximum 500 word answer.)

Question 4 [4 marks]

Write a static Java method, `countProductions`, which counts the number of *productions* in a grammar (see the Amman and Offutt text for a definition of “production”).

The signature for the method should be: `static int countProductions(List<String>)`. The input will be a grammar in BNF or EBNF format (specifically, the notation accepted by the BNF playground), as detailed in question 2.

Your may assume that `java.util.*` and `java.util.regex.*` import statements appear at the top of the source file for your code.

Question 5 [4 marks]

Describe in detail the preconditions and postconditions of the constructor for the `ShopFlightFareCommand` class, justifying your answer. (Max 500 words)

Question 6 [10 marks]

Apply Input Space Partitioning (ISP) to the constructor for the `SegmentSubcommand` constructor, explaining in detail the steps you take and what characteristics and partitions you would use.

You should describe:

- eight characteristics that would be useful for testing the constructor.
- three test cases in detail, including all fixtures, test values and expected values. Include a test ID for each test case, so you can refer to it in question 7.

You need not exhaustively describe all characteristics, partitions, test cases and values that would be needed, but should briefly discuss what more would be needed – beyond the eight characteristics and three test cases you have described – to achieve Base Choice Coverage, and how you would know when it was achieved.

(Max 1000 words)

Question 7 [12 marks]

Write a test class using JUnit 5 called `SegmentSubcommandTest`, which contains `@Test` methods which implement the three test cases you described in question 6. Skeleton code for this class is provided in the supplied Java code, in the "test" directory.

Your test cases should implement all appropriate best practices for unit tests.

Note that to obtain any marks, it is a requirement of the question that:

- you implement three of your question 6 test cases – for each test, include Javadoc documentation providing the test ID from question 6 which it corresponds to. If you implement something else instead, no marks will be awarded.
- your tests properly "Arrange, Act and Assert" your test case. Tests that (for instance) contain no assertions will not be awarded marks.

The code checks available via Moodle can be useful to highlight possible problems in your code, but passing them is not a guarantee of any marks being awarded.

Question 8 [4 marks]

Describe a set of test cases for the `CommandParser.parse()` method which have production coverage of the grammar you specified as an answer to question 1. Explain why it is that your test cases satisfy this coverage criterion.

max 1000 words

Question 10 [4 marks]

Your colleague Oreb has been reading about logic-based testing, and wonders whether it would be a good use of time to assess – once an initial suite of tests is in place – what level of coverage the tests for the GLADIUS system have (e.g. do they have Active Clause Coverage)?

Answer the following question: Would measuring logic-based coverage for the GLADIUS system be a good use of the team's time? Why or why not? Explain your reasoning.

5.1 Extension tasks

You may submit an answer to either of the following questions for 3 bonus marks, awarded at the discretion of the unit coordinator based on the coherence and quality of the answer. These 3 bonus marks cannot take your final mark higher than 50.

- How could mutation testing be used to evaluate the quality of your team's tests, and what empirical evidence exists that mutation testing is a sound approach to evaluating test quality? If you conclude mutation testing is not a sound approach, suggest alternatives, and explain your reasoning. Your answer should include appropriate academic references.
- Write a parser and read–eval–print (REPL) loop for the command syntax described in this project specification, in any language of your choosing that can be compiled and run on an Ubuntu 20.04 distribution running on AMD64 processors. Your REPL loop may return canned responses to commands.

  Commands for building and running your program should be uploaded as an image to the Docker Hub or any other publicly available Docker repository, so that the program can be run using a `docker run` command.

  Your Moodle answer should explain what testing approaches you adopted while writing the program, give a URL on a repository for the source code, and include the details of the `docker run` command needed to run it. (Note: ensure you keep your source code repository private until the marker requests access, as this is an individual project.)