CITS5501/CITS3501 Project 2025 Phase 1

Version:	0.1
Date:	18 August, 2025

PROJECT ADMIN

Project Timeline

- This project contributes **35%** towards your final mark this semester.
- The project has **two deliverables** with 15 marks for Deliverable 1 and 20 marks for Deliverable 2.
- The deadline for **Deliverable 1** is **23:59 Thu 4 September**
- The deadline for **Deliverable 2** is **23:59 Thu 9 October**

Project Rules

Expectations and guidance on the use of AI tools, group formation, project clarifications and marking rubrics, late submission, and academic conduct are provided at CITS5501/3501 FAQs on the unit web pages. For this project you are expected to be familiar with this guidance and comply with the rules.

Project Groups

Students will be allocated to a group of 5 (or 4) students for this project. Each group is assigned a lab facilitator. Groups should meet with their facilitator about once a fortnight during your normal lab time. The facilitator will support you for team forming and project matters and will be the marker for your project submissions.

The group list is available in the CITS5501 ms-teams area

As explained in CITS5501 FAQs individual contributions are taken into account. All group members will be required to complete a Feedback Fruits peer review at the time of the final (phase 2) project submission.

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 CITS5501/3501 Discussion Forum in Moodle using the "project" tag. Any clarifications or amendments that need to be made will be posted by teaching staff in the CITS5501/3501 Discussion Forum

Format and submission of your work

Deliverables 1 and 2 should be submitted as a markdown file that includes links to your group's (private) github repository.

All code and answers to questions should be submitted by making a CITS5501/3501 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 CITS5501/3501 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 following the Oracle guidelines for 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).

Documenting assumptions

If, after posting questions to the CITS5501/3501 Discussion 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 CITS5501/3501 Discussion Forum is *prima facie* not reasonable.

PROJECT SPECIFICATION

Background

Your software development team at KOTO Enterprises is developing a new interface for advanced users of KOTO's travel reservation system, TACHI. The TACHI 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.

Configuration File

KOTO is a global company. They want the TACHI system to be configurable for local contexts. That is, local versions will be run that are applicable only to selected countries (for airports), currencies and airlines.
Your system must include a configuration file called `config.txt` that can be edited by the local offices for their business needs. A simple text-based configuration file is recommended for this project.

TACHI command description

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

TACHI 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 (whether they are meaningful in the context of this application). You should take account of both syntactic and semantic validity in your test cases.

A configuration file is used to specificy which subsets of airports, currencies and airlines are to be used.

Airport codes Airport codes are 3-letter IATA codes for airports – such as SYD (Sydney), PER (Perth), MEL (Melbourne) and CBR (Canberra).

An airport code must be 3 capital letters in the range 'A' to 'Z' inclusive in order to pass syntax validity checks. To be semantically valid it must be a legal IATA airport code **and** one of the country codes listed in your configuration file. Because the list of airports may change over time, current codes should be extracted from a ground truth source. Since access to the official IATA database requires a paid subscription, for this project, you should use Wikipedia's List of international airports by country as the ground truth source.

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 the ISO standard 4217. Only currencies that are valid ISO codes **and** are listed in the configuration file for your local implementation should be accepted in TACHI commands and segment records. A list of up to around 10 selected currencies can be given in the configuration file. You should use this List of ISO 4217 currency codes as your ground truth.

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. Additionally, dates accepted in any TACHI command or segment must be later than or equal to today's date (that is, the date when the system is run).

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. Any 2-letter sequence of capital letters (in the Roman alphabet) is a syntactically valid airline code. To be semantically valid it must be a legal IATA airline code **and** one of the airline codes listed in your configuration file. Because the list of airlines may change over time, you should use extract current codes from a ground truth source.
Since access to the official IATA database requires a paid subscription, for this project, your configuration should use the *IATA 2-digit* airline codes from Wikipedia's List of airline codes as the ground truth source.

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.)

The commands you need to consider in the TACHI system 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 command

The `shop flight fare` 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 fare ORIGIN DESTINATION TRIP_TYPE [LENGTH_OF_STAY] CABIN_TYPE DEPARTURE_DATE`

Both the **`ORIGIN`** and **`DESTINATION`** must be 3-letter IATA airport codes allowed by the local configuration. **`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 request command

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 request`

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 airport 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 request` 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.

TACHI 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 fare` 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 TACHI servers. Other error codes are currently not used.

Supplied files

You are supplied with the Java code skeleton which your team has currently written for the new user interface – see the **`tachi-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 Tachi server.

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

Input Validity: The code includes a skeleton class called `DateTimeChecker` to be used for checking that any date or datetime input in Tachi commands is semantically valid (see above).

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

Note Normally you would delete github branches once you have pulled to the main branch. But for this project please leave your completed branches for review by your facilitator. Your github history will be used to assess your team's software quality management process.

Tasks for Project Deliverable 1 [15 marks]

Task 1 Setting up your group Github Repository [5 marks]

1. **Clone** the provided project repository scaffold from Github at https://github.com/cits5501/project-2025-phase-1-scaffolding. Create in Github a **private** project repository for the members of your group **and your facilitator** (for reveiw and marking).

2. Set up the project by creating a new branch and creating two markdown files in the top directory: `README.md` and `project-phase1-report.md`

3. Make a pull request to commit these files to the master branch. You will notice that some checks are run automatically. Explore the project files to understand the checks that are being made.

4. Modify your `README.md` and `project-phase1-report.md` so that they satisfy the required formatting checks. Once these starter checks are satisfied you will be able to merge your changes to the main branch. A report file `project-phase1-report.pdf` should be generated.

This task will be assessed by reviewing your github repository files and pull requests.

Task 2 Making a Software Quality Plan [7 marks]

In this project you will be designing and writing Java test cases for a command line application called Tachi. You should read carefully the description of this system provided in this project specification.

1. Write a Java code to complete the provided skeleton class, Create a JUnit class `CheckDateTimeStringTest.java` and write JUnit test cases for the methods specified in the skeleton class `CheckDateTimeString.java` that is provided in the `src` directory. Although the Java library already includes date checking classes, for the purposes of this project, you are required to write your own Java checking code and tests.

2. Create at least 2 pull requests for your work on this class each with a **code review by at least 2 team members** (who did not write the code). You *are* ***not*** *required to automate* your pull request checks as a github action, although you may do this if you are familiar with the tools.

3. Briefly describe your software quality assurance plan for this project your `project-phase1-report.md` (max 500 words). Include URL links to your pull requests and any other relevant github components to demonstrate how you are applying these quality checks.

Task 3 Selecting Appropriate Tools [3 marks]

Testing and quality assurance can be expensive activities and just adding more tests may not improve software quality. So don't make your quality assurance tasks too complicated.
Consider the cost of performing the pull rules (e.g. team time for code reviews) and determine where best to spend the time.

1. For this task you are asked to describe one (or more) software quality assurance tasks that you **considered but decided not to use**. Write a brief description of the task considered and your **rationale** for excluding it from your software quailty plan (max 300 words)

Phase 1 Submission

For Phase 1 you should submit the markdown and pdf versions of your `project-phase1-report.md` in the CITS5501/3510 Project submission area in Moodle. *One student only per group should submit the project.* The report should include links to your team's private github repository for the project. Please number the answers in your report using the Task and subtask numbers above e.g. 1.1, 1.2, ... 2.3, 2.4 etc. Remember to ensure your facilitator has access to your github repository.