Project - Skydive Booking System

Original Project Spec below >>

Aims

- Practice how to apply a systematic object-oriented design process
- Gain experience in implementing an object-oriented program with multiple interacting classes
- Learn more about the Java class libraries

Preamble

In this assignment, you will design and implement a prototype system that could serve as the "back-end" of a skydive booking system. Pay careful attention to the information and requirements sections below and make sure you have a complete and sound understanding of them before developing an **object-oriented** design and implementing it. To do this you will need to read the spec multiple times and make your own notes. **The sample input and output files do not form a complete specification**.

Information

In this skydive booking system, skydivers can register, update and cancel jumps. Flights are run by dropzones, and are scheduled with a start and end time and a maximum skydiver load. Each flight facilitates jumps, but the maximum load of skydivers cannot be exceeded due to APF (Australian Parachute Federation) safety regulations.

Jumps are carried out by 1 or more skydivers. There are 3 types of jumps:

- Tandem - a jump where a passenger is attached in front of a "tandem-master" for fun
- Fun-jump - a jump where 1 or more licenced-jumpers jump as a group
- Training - a jump where a skydiver, acting as a trainee, jumps with an instructor to practice skydiving maneouvers and receive feedback on their technique

There are 4 classifications of skydivers:

- Student - jumping for fun as a passenger, or receiving training. Hires gear, so does not repack it.
- Licenced-jumper - qualified skydiver, likely to jump for fun, or could be an instructor if gain further qualifications. Has their own gear, and thus repacks it, unless the passenger in a tandem-jump.
- Instructor - provide training in training jumps. Has their own gear, and thus repacks it, unless the passenger in a tandem-jump.
- Tandem-master - take control in tandem-jumps, and can act as an instructor (they have all instructor qualifications). Has their own gear, and thus repacks it, unless the passenger in a tandem-jump.

All tandem-masters are qualified as instructors and can fulfill the duties of an instructor. From this point onwards in this specification, "instructor" will refer to both instructors and tandem-masters unless specified otherwise.

All tandem-masters and instructors are qualified as licenced-jumpers and can act as such. From this point onwards in this specification, "licenced-jumper" will refer to licenced-jumpers, instructors and tandem-masters

unless specified otherwise.

All skydivers can act as a student (they need additional training to progress, after all).

Parachutes require 10 minutes to repack immediately after the jump. Thus, licenced-jumpers cannot jump for 10 minutes immediately after a jump. The only exception to this is that if a licenced-jumper was the passenger in a tandem jump, they would not be required to repack gear and thus can immediately jump again. Students are never required to repack gear and thus do not have to wait for repacking to finish.

Tandem jumps have 2 skydivers - a Tandem-master controlling the jump, and a passenger (who is a skydiver of any classification). They require a 5-minute briefing immediately before the jump which prevents all participating skydivers from jumping (or completing other briefs, debriefs, or repacking).

Training jumps have 2 skydivers - 1 trainee (of any classification), and 1 instructor. They require an additional 15-minute debrief immediately after the jump which prevents both participating skydivers from jumping (or completing other briefs, debriefs, or repacking). This occurs before the 10-minutes of repacking, which must be completed by licenced jumpers.

In this assignment, you will have to be able to generate "jump-runs". This provides the order in which fun-jump groups/training jumps/tandem-jumps will exit the plane in a flight (from first to exit to last), and is calculated to maximise horizontal safety between groups, to increase safety. The order to exit the plane is (from first to last):

- Largest to smallest fun-jump groups (including individuals)
- Largest to smallest training-jump groups
- Tandem-jumps

For this assignment, if 2 jumps have the same classification and group size, the first jump to be booked will exit the plane first (for the purposes of this, a jump resulting from a `change` command will be considered a newly-booked jump).

Assignment Requirements

In this skydive booking system, skydivers can make, change and cancel jumps. A jump request has a named identifier and is for a type of jump ("fun", "training", or "tandem"), with an earliest start time. If the jump is a fun-jump, the names of all fun-jumpers are specified. For training/tandem jumps, the trainee/passenger names are specified.

The preferred starttime provided in the booking is the time the skydiver/s will be available to start a brief, or jump if no briefing is required. If the preferred start time cannot be fulfilled, the next flight which can fit the jump on the day will be booked, otherwise the request will fail if no flight can fit the jump on the required day.

Assessment will be based on the design of your program in addition to correctness. You should submit at least a full UML class diagram used for the design of your program, i.e. not generated from code afterwards. The UML diagram should include all details of a full UML diagram - including correct cardinalities, relationship annotations, and private and public members and functions. You may exclude simple getter and setter functions for variables from your UML diagram (functions immediately assigning a provided value to a field, or immediately returning a field).

You should also submit at least 3 sets of JSON input and output test files you used to test your program (in the same fashion as the `sample_input.json` and `sample_output.json` files have been provided).

The implementation should input and output data in the JSON format. It will read from STDIN (`System.in` in Java) and output to STDOUT (`System.out` in Java). The input will be a series of JSON objects, each containing a single command (on its own line). After reading in a JSON object, the implementation will immediately process that command; i.e. it cannot wait for all commands to be input before acting on them. You can assume `skydiver` and `flight` commands will precede all other commands, but beyond that there are no guarantees of the order or quantity of each command. All `skydiver`, `flight`, `cancel` and `jump-run` commands provided to the system will be valid. An unseen dropzone should be created when first seen within either a `skydiver` or `flight` command. Dropzones are uniquely identified by their dropzone name (the "dropzone" field) - so dropzones with the same dropzone name must be the same dropzone.

The commands are as follows (the text in italics is what will vary):

- Specify that dropzone *dropzone* has a flight with id *id* from *starttime* to *endtime*, with a maximum skydiver load of *maxload* skydivers.

  > { "command": "flight", "id": *id*, "dropzone": *dropzone*, "starttime": *starttime*, "endtime": *endtime*, "maxload": *maxload* }

- Specify that skydiver *skydiver* has licence *licence* (for either "student", or "licenced-jumper").

  > { "command": "skydiver", "skydiver": *skydiver*, "licence": *licence*}

- Specify that skydiver *skydiver* has licence *licence* (for either "instructor", or "tandem-master"), and home dropzone *dropzone*.

  > { "command": "skydiver", "skydiver": *skydiver*, "licence": *licence*, "dropzone": dropzone}

- Request to create jump *id* from earliest time *starttime*, of type "fun" (fun-jump skydive), with a list of skydivers [*skydiver*].

  > { "command": "request", "type": "fun", "id": *id*, "starttime": *starttime*, "skydivers": [*skydiver*] }

- Request to create jump *id* from earliest time *starttime*, of type "tandem" (tandem skydive), for passenger *skydiver*.

  > { "command": "request", "type": "tandem", "id": *id*, "starttime": *starttime*, "passenger": *skydiver* }

- Request to create jump *id* from earliest time *starttime*, of type "training" (training skydive), for trainee *skydiver*.

  > { "command": "request", "type": "training", "id": *id*, "starttime": *starttime*, "trainee": *skydiver* }

- Change existing jump *id* to be of type "fun", from earliest time *startime*, with a list of skydivers [*skydiver*].

  > { "command": "change", "type": "fun", "id": *id*, "starttime": *starttime*, "skydivers": [*skydiver*] }

- Change existing jump *id* to be of type "tandem", from earliest time *starttime*, for passenger *skydiver*.

  > { "command": "change", "type": "tandem", "id": *id*, "starttime": *starttime*, "passenger": *skydiver* }

- Change existing jump *id* to be of type "training", from earliest time *starttime*, for trainee *skydiver*.

  > { "command": "change", "type": "training", "id": *id*, "starttime": *starttime*, "trainee": *skydiver* }

- Cancel jump *id* and free up skydivers

  > { "command": "cancel", "id": *id* }

- Generate the jump-run for a flight with id *id*

  > { "command": "jump-run", "id": *id* }

Flight times can overlap - but no flights or jumps will be registered with the same id (this includes that a flight will not have the same id as a jump). The `skydiver` command will not be run more than once for any particular skydiver name (the "skydiver" field). Skydivers cannot participate in more than 1 jump per flight, and participation in a jump excludes them from participating in a jump (and associated briefings/debriefings/repacking) for the duration of briefings, the jump, debriefings, and repacking, as applicable to that jump for that skydiver.

To remove any ambiguity, **all** jump requests and changes are fulfilled as follows: each flight by earliest to latest starttime on the chosen day is checked (starting at the earliest flight allowed by the start-time, after taking into account any briefing times) to determine whether it can allow all participating skydivers, and provide trainers and tandem-masters from the dropzone providing the flight (taking into account the availability of all skydivers). If multiple flights (including across dropzones) have the same flight starttime, the dropzone with the most total skydiver vacancies (i.e. unbooked capacity in the flights) across all flights is prioritized. If dropzones have the same number of skydiver vacancies across all flights, we prioritize flights which were registered earlier. The first flight in this order which can allow the jump will book the jump. If none can allow the jump, no changes occur, and rejection is output to the terminal as specified in this spec. The output of a successful request/change should list the dropzone and flight (flight id) hosting the jump. Do not try to fulfil requests by reassigning any skydivers or jumps between flights or dropzones to create space etc.

Allocation of instructors/tandem-masters to a jump is always in favour of those with fewer jumps on the day (including fewer jumps as a trainee and passenger). Instructors and tandem masters can only act as tandem-masters and instructors at their home dropzone (the dropzone in the "dropzone" field for their registration). For instructors/tandem-masters with the same number of jumps on a day, priority goes to those registered first in the system.

For the `jump-run` command, output an array containing the jumps from the first to exit the plane, to the last to exit (as described in the "Information" section of this specification). Tandem jumps should list the passenger in the "passenger" field, and the tandem master in the "jump-master" field. Training jumps should list the trainee in the "trainee" field, and the instructor in the "instructor" field. Fun-jumps should list the skydivers (in lexicographic order) in an array in the "skydivers" field.

In this system, times of jumps are exclusive (i.e. a skydiver starting/ending a jump with a particular time does not prevent the skydiver respectively ending/starting a jump at the same time. For example, a skydiver can finish repacking for a fun-jump at midday and start another fun-jump precisely at midday). However, overlap between jump times is still not allowed (i.e. the same skydiver would not be allowed to finish repacking at midday, but start their next jump at 11:59am on the same day).

No invalid input will be provided by the tests - so you are not required to handle any invalid input such as having end times before start times or malformed input. All flights will start and end on the same day.

For passing tests - you must pass the entirety of the test to receive marks on a correctness test - there will be no partial marks.

A **successful** change matches the effect of cancelling a request and making a new request with the new requirements. However, an **unsuccessful** attempt to make a change should have no effects on the state of the system. The fields in a `change` command are the same type (the value associated with the same key across these commands will represent the same information, and have the same data type) as in a `request` command, except that the values for the "command" field are different.

Additional Requirements

You must ensure that your entire Gitlab repository uses less than 5MB of space. This includes the most recent 10 commits before a submission.

Implementation

Starter code has been provided in your repository for this assignment, available here (replace z5555555 with your own zID):

https://gitlab.cse.unsw.edu.au/z5555555/20T3-cs2511-ass1

Create all your Java source files in the `unsw.skydiving` package. You may create subpackages if you wish, but this is not required. The main Java file is `SkydiveBookingSystem.java`. Do not rename this class. The starter code includes the JSON-java library and shows how to use it to read and write JSON formatted data. Other than this, you are **not allowed to use any third party libraries**.

For machine marking, the output will be directly compared to the expected output, so do not print out any extra debugging information or include extra fields in your JSON objects. You can assume that identifiers and names are unique for different flights, skydivers and jumps (e.g. there won't be two flights under the flight id `flight1`; if `flight1` is referenced in multiple places it refers to the same flight). Similarly, datetimes will be in the ISO-8601 format `uuuu-MM-dd'T'HH:mm` (e.g. `2007-12-03T10:15`). This is the format output by `LocalDateTime.toString()`.

All skydiver, jump, dropzone and flight identifiers/names will be composed only of ASCII characters. The input to this system is trusted, so you can assume it will not be malformed, fields will be of the right type, datetimes will be valid, end datetimes do not come before start datetimes, etc.

Example

This is a concrete example input demonstrating the commands supported:

```
{ "command": "flight", "id": "flight1", "dropzone": "Picton", "starttime": "2007-12-02T10:15", "endtime": "2007-12-02T10:35", "maxload": 4 }
{ "command": "flight", "id": "flight2", "dropzone": "Picton", "starttime": "2007-12-02T10:25", "endtime": "2007-12-02T10:40", "maxload": 7 }
{ "command": "flight", "id": "flight3", "dropzone": "Picton", "starttime": "2007-12-02T10:40", "endtime": "2007-12-02T11:00", "maxload": 5 }
```

```
{ "command": "flight", "id": "flight4", "dropzone": "Picton", "starttime": "2007-12-03T10:15", "endtime": "2007-12-03T10:35", "maxload": 4 }
{ "command": "flight", "id": "flight5", "dropzone": "Picton", "starttime": "2007-12-03T10:25", "endtime": "2007-12-03T10:40", "maxload": 7 }
{ "command": "flight", "id": "flight6", "dropzone": "Picton", "starttime": "2007-12-03T10:40", "endtime": "2007-12-03T11:00", "maxload": 5 }
{ "command": "flight", "id": "flight7", "dropzone": "Moruya", "starttime": "2007-12-04T17:00", "endtime": "2007-12-04T17:10", "maxload": 5 }
{ "command": "flight", "id": "flight8", "dropzone": "Picton", "starttime": "2007-12-04T16:00", "endtime": "2007-12-04T16:10", "maxload": 5 }

{ "command": "skydiver", "skydiver": "Matt", "licence": "licenced-jumper"}
{ "command": "skydiver", "skydiver": "Bruno", "licence": "student"}
{ "command": "skydiver", "skydiver": "Ted", "licence": "licenced-jumper"}
{ "command": "skydiver", "skydiver": "Alice in Wonderland", "licence": "tandem-master", "dropzone": "Picton"}
{ "command": "skydiver", "skydiver": "Ashesh", "licence": "tandem-master", "dropzone": "Picton"}
{ "command": "skydiver", "skydiver": "Apple Banana Coconut", "licence": "tandem-master", "dropzone": "Picton"}
{ "command": "skydiver", "skydiver": "Borat", "licence": "instructor", "dropzone": "Picton"}

{ "command": "request", "type": "tandem", "id": "forgot to arrive in time, in Wonderland, need 5 minute briefing", "starttime": "2007-12-03T10:40", "passenger": "Alice in Wonderland" }
{ "command": "request", "type": "tandem", "id": "Alice arrived just in time for tandem with Ashesh", "starttime": "2007-12-03T10:35", "passenger": "Alice in Wonderland" }

{ "command": "request", "type": "training", "id": "Bruno arrives for training, but there are no instructors!", "starttime": "2007-12-04T17:00", "trainee": "Bruno" }
{ "command": "request", "type": "training", "id": "Bruno arrives for training with Alice in Wonderland", "starttime": "2007-12-04T16:00", "trainee": "Bruno" }

{ "command": "request", "type": "fun", "id": "Matt, Ashesh and Borat go out to do some sick jumps, but Ashesh unfortunately has tandem booked with Alice, he wont be able to pack in time!", "starttime": "2007-12-03T10:15", "skydivers": ["Matt", "Ashesh", "Borat"] }

{ "command": "request", "type": "fun", "id": "Matt, Ashesh and Borat go out to do some sick jumps", "starttime": "2007-12-02T10:15", "skydivers": ["Matt", "Ashesh", "Borat"] }

{ "command": "request", "type": "fun", "id": "Matt can't do the next jump because he is busy repacking his gear. He's very annoyed!", "starttime": "2007-12-02T10:40", "skydivers": ["Matt"] }

{ "command": "change", "type": "fun", "id": "Matt, Ashesh and Borat go out to do some sick jumps", "starttime": "2007-12-02T10:15", "skydivers": ["Matt", "Ashesh", "Borat", "Ted"] }

{ "command": "change", "type": "fun", "id": "Matt, Ashesh and Borat go out to do some sick jumps", "starttime": "2007-12-02T10:15", "skydivers": ["Matt", "Ashesh",
```

```
"Borat", "Ted", "Apple Banana Coconut"] }

{ "command": "request", "type": "fun", "id": "Matt throws in an extra jump before
doing a long night of marking exam papers!", "starttime": "2007-12-04T17:00",
"skydivers": ["Matt"] }

{ "command": "cancel", "id": "Matt throws in an extra jump before doing a long
night of marking exam papers!" }

{ "command": "request", "type": "fun", "id": "Matt changes his mind and wants
another jump", "starttime": "2007-12-03T10:40", "skydivers": ["Matt"] }

{ "command": "jump-run", "id": "flight1" }
{ "command": "jump-run", "id": "flight2" }
{ "command": "jump-run", "id": "flight3" }
{ "command": "jump-run", "id": "flight4" }
{ "command": "jump-run", "id": "flight5" }
{ "command": "jump-run", "id": "flight6" }
{ "command": "jump-run", "id": "flight7" }
{ "command": "jump-run", "id": "flight8" }
```

What is happening in the above commands is usually explained by the `id` for the requests. For the 2 `change` command examples in the above example:

- For the first example of the `change` command above, it tries changing with the same details, but with a 4th Skydiver "Ted" (succeeding).
- For the second example of the `change` command above, it tries changing with the same details, but with a 5th Skydiver "Apply Banana Coconut". Since there are not enough spots on flight 1, it is scheduled to flight 2 (the first flight on the same day at or after the preferred starttime which is able to accommodate the jump).

Of these commands, `request`, `change`, and `jump-run` will produce output. The other commands do not. Rejected requests or changes should result in outputting a JSON dictionary displaying that the status has been rejected (as below). Successful requests or changes should result in outputting a JSON dictionary reporting status "success", the flight id, and the dropzone name (as below).

Inputting the above input example should yield the following (ordering of JSON dictionary fields and indentation may differ):

```
{"status": "rejected"}
{
  "flight": "flight6",
  "dropzone": "Picton",
  "status": "success"
}
{"status": "rejected"}
{
  "flight": "flight8",
  "dropzone": "Picton",
  "status": "success"
```

```
}
{"status": "rejected"}
{
  "flight": "flight1",
  "dropzone": "Picton",
  "status": "success"
}
{"status": "rejected"}
{
  "flight": "flight1",
  "dropzone": "Picton",
  "status": "success"
}
{
  "flight": "flight2",
  "dropzone": "Picton",
  "status": "success"
}
{
  "flight": "flight7",
  "dropzone": "Moruya",
  "status": "success"
}
{
  "flight": "flight6",
  "dropzone": "Picton",
  "status": "success"
}
[]
[{"skydivers": [
  "Apple Banana Coconut",
  "Ashesh",
  "Borat",
  "Matt",
  "Ted"
]}]
[]
[]
[]
[
  {"skydivers": ["Matt"]},
  {
    "passenger": "Alice in Wonderland",
    "jump-master": "Ashesh"
  }
]
[]
[{
  "instructor": "Alice in Wonderland",
  "trainee": "Bruno"
}]
```

Note that all commands produce a JSON object as output, except for `jump-run` that produces a JSON array.

For commands that do not always succeed, the `status` field indicates whether the result was successful. If a request or change cannot be fulfilled, the status should be `rejected`. In the case of such a rejection, no jumps should be added, changed or deleted. You can assume `change` and `cancel` have identifiers for existing jumps, `jump-run` has a valid flight, and `skydiver` and `flight` are not used to add a skydiver or flight that has already been added.

Hints

- Focus on the requirements as given. A solution that tries to satisfy requirements that weren't given is not necessarily a better solution.
- The only data structures you will need are lists. Structures like HashMaps are neither necessary nor improve your solution. They may reduce the quality of your solution if you are not careful by mixing up the required output order.
- The JSON-Java library is intended for serialisation, not as an alternative to Java collections.
- Ensure you implement correct roll-back functionality in case a change fails/you otherwise implement some mechanism to ensure failed changes don't change the system state.
- A **successful** change matches the effect of cancelling a request and making a new request with the new requirements. Thus, ensure that overlap between the jump being changed and the appropriate replacement jump wont prevent approval of the change.
- For command output, ordering of **fields and indentation** are allowed to differ in your submission. This is likely to occur because JSONObjects may not preserve all parts of the original order.
- Skydivers and flights will not be created twice.
- You may wish to consider using ArrayLists.
- A request for a jump can only reserve 1 jump (not mix and match jumps on different dates).
- Assume all skydivers names/flights/dropzones/etc. are case-sensitive.
- In a training jump, if the trainee is only a student, they will not repack their gear, but if they are a licenced-jumper/instructor/tandem-master, they will repack their gear.
- In a tandem-jump, the passenger never repacks gear, no matter their skydiving classification.
- All skydivers count towards the maximum skydiver load for a flight - including instructors/tandem-masters.
- Jump requests must take into account the availabilities of all skydivers participating, including instructors/tandem-masters.
- Change commands can change the type of the jump (as long as there is availability) - e.g. from tandem to fun-jump.
- Requests will not be repeated (i.e. there will be no 2 request commands with the same "id" value).
- Ensure that if an instructor or tandem-master applies for training or a tandem jump as a trainee/passenger, they are not also the instructor/jump-master for that jump.

Submission

You should ensure the following are in your GitLab repository:

- All your .java source files
- A .pdf file containing your design documents (a UML class diagram and, optionally, other diagrams necessary to understand your design)
- A series of .json files (at least three) that you have used as input files to test your system, and the corresponding .json output files (call these input1.json, output1.json, input2.json, output2.json, etc.)

Submit the contents of your repository with:

```
2511 submit ass1
```

The last submission as of the date of assessment will be marked. You may submit as many times as you wish.

The above submit command **MUST** be run from a CSE machine remotely or in-lab. It submits the contents of your Gitlab repository. You must also read the submit command output, and input the required responses.

A simple check will be done by the submit command to ensure your code compiles and works correctly with the sample input. If your solution does not work for the sample input you can still submit, but you should not expect high marks.

Important Points

- You must read this assignment specification thoroughly. Failure to incorporate any elements may result in a loss of marks.
- You should assume that your JSON output is consumed by another program, and therefore your output MUST satisfy all the requirements. No human marking will be applied in place of the automarked correctness marks under any circumstances - thus it is your responsibility to ensure your program satisfies the requirements 100%. We will not accept arguments similar to "my output is almost the same as the required output so I should get some marks".
- Under no circumstances will any modifications be made by your tutor to your code to fix it during marking (even if the problem is small and loss of marks is severe). You are responsible for ensuring it is properly tested and passes the dryrun. You should plan for the possibility that you will need to spend additional time to solve configuration issues - particularly by having it ready by your week 4 tutorial, in case you need to seek help from a tutor.
- Be careful to follow the specification on ordering. If you output the incorrect order you are likely to lose many marks for correctness.
- The dryrun intentionally does not test every requirement of the specification. You should develop additional tests from the spec to check the correctness of your submission. No more tests will be provided by the course staff.
- If there are modifications made to the assignment specification to clarify ambiguities, these will be announced via webcms3, and you will be expected to incorporate them into your program (even after

commencement of the assignment).

- You should also take into account that last-minute-submissions may encounter glitches or slower submission due to many students attempting to submit at once. You should submit early to avoid this. Failure to submit in time due to a technical glitch from submitting just before the deadline will result in you receiving a late penalty.
- Any commits in your repository that were made after a submission will not be considered when marking.
- The JSON test files must not be the same as the tests already provided - to receive marks you must develop new tests from the specification.
- Your submission correctness automarking will use Java 11, as installed on CSE machines.
- You should leave the directory structure and package labels as originally labelled in the starter code - the automarking expects that structure.

Assessment

Marks for this assignment are allocated as follows:

- Correctness (automarked): 7 marks
- Design (marking of submission by tutor): 7 marks
- Style (marking of submission by tutor): 1 mark

Late penalty: 1.5 marks per day (1.5 of 15 marks, i.e. 10% of the assignment) or part-day late off the maximum mark obtainable for up to 5 (calendar) days after the due date (after which your mark will be reduced to 0).

Assessment Criteria

- Correctness: Automatically assessed on a series of input tests
- Design: Adherence to object-oriented design principles, clarity of UML diagrams and conformance of UML diagrams to code, JSON test files
- Programming style: Adherence to standard Java programming style, understandable class and variable names, adequate Javadoc and comments