Mcq - Model

MCQ Java application with GUI, model part.

In this part, the goal is to implement the **model** from:

- UML class diagrams, located in `./part1-Model/uml` directory.
- Generated *Javadoc*, located in `./part1-Model/docs` directory.
  - **Javadoc comments are not to be rewritten**.
- Following guidelines.

package structure is the following:

- `fr.iutvalence.info.but.s2_01.mcq.launchers.validation` contains applications used to validate features,
- `fr.iutvalence.info.but.s2_01.mcq.model.core` contains core classes,
- `fr.iutvalence.info.but.s2_01.mcq.model.exceptions` contains exception classes.

Table of contents

Implementation guidelines

The next section details tasks to be performed.

1. Project configuration

Doing it

1. **Configure project** so that:
   - `/part1-Model/src` is considered as *source*,
   - any other directory (especially `/part0-Preamble/src`) is considered as *excluded*.
2. **Copy** `Main` from `/part0-Preamble/src/.../preamble` to `/part1-Modem/src/.../validation` (where all other validation applications will be written).

Checking it

Check that project builds successfully and that `Main` execution is as expected.

Committing/Pushing it

- **Commit** changes with `1-Model-Configuration` as message brief.
- **Push** immediately.
- **Check** that remote repository has been updated.

2. Question model

(See **UML class diagram** in `/part1-Model/uml/Question-ClassDiagram.png` .)

`Question` class defines a *question* by:

- a **text**,
  - e.g. `"What is the answer to life, universe and everything?"` .
- an **array of answers**,
  - e.g. `"42", "32768", "There is no answer", "Kamoulox"` .
- a **correct answer id** (ids start at 0, and are considered as array indexes).
  - e.g. `0` .

Calling `toString` method on an object created with the values previously given as examples returns a description like:

```
? -> What is the answer to life, universe and everything?
0 -> (o) 42
1 -> (x) 32768
2 -> (x) There is no answer
3 -> (x) Kamoulox
```

`QuestionMain` application validates `Question` class behaviour by creating an instance and writing its representation to standard output.

Doing it

With the help of UML class diagrams and *Javadoc*,

1. **Complete** `Question` source code.
2. **Complete** `QuestionMain` application source code, considering that it creates the `Question` object given as example and writes its representation on standard output.

Checking it

Check that `QuestionMain` execution output is as expected (see above).

Committing/Pushing it

- **Commit** changes with `1-Model-Question` as message brief.
  - If checking step has failed, give details about issues.
- **Push** immediately.
- **Check** that remote repository has been updated.

3. Questionnaire model

(See **UML class diagram** in `/part1-Model/uml/Questionnaire-ClassDiagram.png` .)

`Questionnaire` class defines a *questionnaire* by:

- an **author name**,
  - e.g. `"Myself"`
- a **title**,
  - e.g. `"My questionnaire"`
- an **array of questions**.

Calling `toString` method, on an object created with values previously given as examples, returns a description like:

```
Title:  My questionnaire
Author: Myself
2 question(s)
Question 0
? -> What is the answer to life, universe and everything?
0 -> (o) 42
1 -> (x) 32768
2 -> (x) There is no answer
3 -> (x) Kamoulox
Question 1
? -> Another question?
0 -> (x) I don't know
1 -> (o) No
2 -> (x) For sure
3 -> (x) Maybe
```

`QuestionnaireId` class defines a *unique questionnaire identifier* made of **author name and title**.

`QuestionnaireMain` application validates `Questionnaire` class behaviour by creating an instance and writing its representation to standard output.

Doing it

With the help of UML class diagrams and *Javadoc*,

1. **Complete** `QuestionnaireId` source code.
2. **Complete** `Questionnaire` source code.

3. **Complete** `QuestionnaireMain` application source code, considering that it creates a `Questionnaire` object and writes its representation on standard output (expected output is the one given above).

Checking it

Check that `QuestionnaireMain` execution output is as expected (see above).

Committing/Pushing it

- **Commit** changes with `1-Model-Questionnaire` as message brief.
  - If checking step has failed, give details about issues.
- **Push** immediately.
- **Check** that remote repository has been updated.

4. Submission model

(See **UML class diagram** in `/part1-Model/uml/Submission-ClassDiagram.png` .)

`Submission` class defines a *submission* (a questionnaire filled by someone) by:

- a **filler name**,
  - e.g. `"Anonymous filler"`
- a **questionnaire id**,
- an **array of answer ids**.

Calling `toString` method on an object returns a description like:

```
Filler: Anonymous filler
Questionnaire: Myself#My questionnaire
Question 0: 1
Question 1: 3
```

`SubmissionId` class defines a *unique submission identifier* made of filler name and questionnaire id.

`SubmissionMain` application validates `Submission` class behaviour by creating an instance and writing its representation on standard output.

Doing it

With the help of UML class diagrams and *Javadoc*,

1. **Complete** `SubmissionId` source code.
2. **Complete** `Submission` source code.
3. **Complete** `SubmissionMain` application source code, considering that it creates a `Submission` object and writes its representation on standard output (expected output is the one given above).

Checking it

Check that `SubmissionMain` execution output is as expected (see above).

Committing/Pushing it

- **Commit** changes with `1-Model-Submission` as message brief.
  - If checking step has failed, give details about issues.
- **Push** immediately.
- **Check** that remote repository has been updated.

5. Questionnaire/submission manager

(See **UML class diagram** in `/part1-Model/uml/McqManager-ClassDiagram.png` .)

`McqManager` class allows to manipulate *collections of questionnaires and submissions* (create, retrieve, update, delete).

Doing it

With the help of UML class diagrams and *Javadoc*,

1. **Complete** `McqManager` source code.
2. **Complete** `McqManagerMain` application source code, considering guidelines given in source code.

Checking it

Check that `McqManagerMain` execution output is as below:

```
0 questionnaire(s)
Adding a questionnaire
1 questionnaire(s)
Title:  My questionnaire
Author: Myself
2 question(s)
Question 0
? -> What is the answer to life, universe and everything?
0 -> (o) 42
1 -> (x) 32768
2 -> (x) There is no answer
3 -> (x) Kamoulox
Question 1
? -> Another question?
0 -> (x) I don't know
1 -> (o) No
2 -> (x) For sure
3 -> (x) Maybe
0 submission(s)
Adding a submission
1 submission(s)
Filler: A filler
```

```
Questionnaire: Myself#My questionnaire
Question 0: 1
Question 1: 3
Removing a questionnaire
true
0 questionnaire(s)
Removing a submission
true
0 submission(s)
```

Committing/Pushing it

- **Commit** changes with `1-Model-McqManager` as message brief.
  - If checking step has failed, give details about issues
- **Push** immediately.
- **Check** that remote repository has been updated.

6. question/questionnaire/submission builders

(See **UML class diagram** in `/part1-Model/uml/Builders-ClassDiagram.png` .)

`QuestionBuilder` , `QuestionnaireBuilder` and `SubmissionBuilder` classes allow to respectively build step by step `Question` , `Questionnaire` and `Submission` instances.

Each builder:

- defines **attributes that mimic those of the object** (e.g. `question` , `answers` and `correctAnswerId` for `Question`),
- provides `set` / `add` / `update` / `remove` methods for these attributes,
  - some of these methods check that operations are valid, and throw exceptions if not.
- provides a `get` method that **calls constructor and returns instance**, checking that object creation is valid and throwing exceptions if not.

Warning:

- exceptions have to be defined in `fr.iutvalence.info.but.s2_01.mcq.launchers.model.exceptions` package.
- some exceptions are *unchecked* (*runtime*) exceptions (see Javadoc).

Doing it, checking it, committing/pushing it

With the help of UML class diagrams and *Javadoc*,

1. **Complete** `QuestionBuilder` source code, considering the following possible error messages (associated to exceptions):
   - `"Answer is empty"` ,
   - `"Question text is missing"` ,
   - `"Question has no answer"` ,

- `"Correct answer id is missing"` ,
- `"Correct answer id is out of bounds"` .

2. **Complete** `QuestionBuilderMain` application source code by creating the same `Question` object than in `QuestionMain` application but by using a `QuestionBuilder` .
   - *N.B. if* `getQuestion` *throws an exception, then error message is written on standard output and application shuts down.*
3. **Check** that `QuestionBuilderMain` execution output is the same as `QuestionMain` .
4. **Commit** changes with `1-Model-QuestionBuilder` as message brief (and details if checking failed), push immediately, and check that remote repository has been updated.
5. **Complete** `QuestionnaireBuilder` source code, considering the following possible errors message (associated to exceptions):
   - `"Questionnaire title is missing"` ,
   - `"Questionnaire author name is missing"` ,
   - `"Questionnaire has no question"` .
6. **Complete** `QuestionnaireBuilderMain` application source code by creating the same `Questionnaire` object as in `QuestionnaireMain` application but by using a `QuestionnaireBuilder` .
   - *N.B. if* `getQuestionnaire` *throws an exception, then error message is written on standard output and application shuts down.*
7. **Check** that `QuestionnaireBuilderMain` execution output is the same as `QuestionnaireMain` .
8. **Commit** changes with `1-Model-QuestionnaireBuilder` as message brief (and details if checking failed), push immediately, and check that remote repository has been updated.
9. **Complete** `SubmissionBuilder` source code, considering the following possible errors message (associated to exceptions):
   - `"Submission filler name is missing"` ,
   - `"Submission questionnaire id is missing"` ,
   - `"Submission has no answer"` .
10. **Complete** `SubmissionBuilderMain` application source code by creating the same `Submission` object as in `SubmissionMain` application but by using a `SubmissionBuilder` .
    - *N.B. if* `getSubmission` *throws an exception, then error message is written on standard output and application shuts down.*
11. **Check** that `SubmissionBuilderMain` execution output is the same as `SubmissionMain` .
12. **Commit** changes with `1-Model-SubmissionBuilder` as message brief (and details if checking failed), push immediately, and check that remote repository has been updated.