Curtin University — Discipline of Computing

Software Engineering Concepts (COMP3003)

Semester 2, 2018

Assignment

Date: Thursday 25 October, 23:59:59 AWST / 21:30:00 IST

Weight: 25% of the unit mark.

Your task is to develop a GUI-based, plugin-based quiz application, which asks the user a sequence of questions, with timeouts, and records the number they score correctly on. You must also develop documentation explaining how to write the plugins.

1 GUI Programming

This assignment relies on developing a GUI. I strongly recommend using either JavaFX or Swing for this purpose. The UI doesn't have to "look nice", and you won't get any marks for aesthetics, provided it functions as required.

I am happy to provide assistance as needed in working with GUI windows and widgets. You won't need any particularly advanced ones: labels, radio buttons, text fields, and ordinary buttons, and containers (boxes/panels) should basically take care of it.

While there are tools that can help you design GUIs, be wary of them in this case. They may actually lead you down the wrong path, because a lot of your GUI will need to be updated dynamically and programmatically at runtime (as a result of the plugin architecture).

2 Plugin Architecture

The quiz application must have *two types* of plugins:

(a) A "quiz plugin" contains the structure of a quiz – all the questions and answers, and the sequence of questions. Each quiz plugin represents a complete quiz on a given subject area.

You must include at least one quiz plugin as part of your project, and it should demonstrate the full range of possible quiz plugin functionality.

(b) A "question-type plugin" arranges for a particular kind of question to be displayed on the screen. As part of your project, you must include at least two question-type plugins, one to implement each of the following question types:

- A multichoice question, with 2 or more possible answers and GUI radio buttons to select between them;
- A short answer question, with a simple text input field.

In theory, someone else could design additional question-type plugins to add more question types, and additional quiz plugins to add different quizzes. Your design must make this possible, *without* needing to update any of the main application code.

Obviously, the quiz plugins and question-type plugins will need to interact:

- Quiz plugins need a (reasonably simple) way of loading question-type plugins. This loading mechanism must be provided by the main application.
- When a quiz plugin creates a particular question, it must be able to specify what kind of question-type plugin will handle it. It must also be able to provide information specific to the question type.

The following is an example of what a quiz plugin may look like:

```
public class CricketQuiz implements Quiz
{
    @Override
    public void runQuiz(QuestionTypeLoader loader)
    {
        // Load question-type plugins.
        MultiChoice mc = (MultiChoice) loader.load("multichoiceplugin");
        ShortAnswer sa = (ShortAnswer) loader.load("shortanswerplugin");

        // Create a short-answer question.
        Question q1 = sa.makeShortAnswerQuestion(
            "How many overs are usually played in contemporary "
                + "'one day international' matches?", // Question
            "50"); // Answer

        // Create a multi-choice question.
        Question q2 = mc.makeMultiChoiceQuestion(
            "Which of the following is NOT a valid fielding position?"
            new String[] {
                "Silly mid off", "Short leg", "Catcher",
                "Backward point", "Cow corner" }, // Possible answers
            2); // Index of correct answer ("Catcher")

        // Actually display each question and wait for an answer.
        q1.invoke(...);
        q2.invoke(...);
    }
}
```

Note: I have separated out *making* a question from *invoking* it. This is not crucial to the requirements though – you could perform both in one step.

The overall required sequence of events is as follows:

- The user opens your main application and selects a particular quiz plugin. (This can be as simple as asking the user to enter a filename.)
- The main application loads the quiz plugin, and executes `runQuiz()` (or equivalent).
- As shown above, the quiz plugin (using a mechanism provided by the main application) loads the question-type plugins.
- As shown above, the quiz plugin asks the question-type plugin(s) to create questions.
- Each question is "invoked", causing it to be displayed on-screen. All three major components of the application must be involved here. The quiz plugin invokes the question. The question-type plugin determines how to display it. The main application itself needs to be involved too for reasons that will become clear in the next section. Some careful design work is required here!
- The user answers the question (by selecting a radio button, entering some text, or using whatever other UI controls the question-type plugin has provided) and presses a "submit" button.

  (You may choose to disable the submit button until the user has actually entered an answer, but this is optional.)
- The relevant question-type plugin determines whether the answer is correct or incorrect, and displays the result.
- The user now presses a "next" button to proceed to the next question.
- As implied by the above code, the `runQuiz()` method will return once the last question is done. The application then displays the user's score; i.e. the number of correctly answered questions, out of the total number of questions.

3 Other Functionality

3.1 Timeouts

The quiz plugin must be able to specify a timeout (measured in seconds) for each question. If a timeout occurs (i.e. the user takes more than x seconds to answer), the user's current answer will be automatically submitted.

The application must display the number of seconds remaining to answer the question (rounded up to the nearest whole second). Optionally, your application can *also* show this information graphically using a progress bar.

The user may in fact have already entered the correct answer when a timeout occurs, and if so this is simply regarded as a correct answer. If no answer or an otherwise incorrect answer is entered, and then a timeout occurs, the user simply gets the question wrong.

The quiz plugin code might have the following:

```
q1.invoke(30); // The user has 30 seconds to answer Q1.
q2.invoke();   // The user has unlimited time to answer Q2.
```

The period being timed starts from when the question is first displayed to the user, and stops when they submit or when a timeout occurs. The user always has unlimited time to view the result for each question.

3.2 Returning Future Results

The quiz plugin should be able to query the result of a question – i.e. whether the user gets it right or wrong – via a `Future<Boolean>` object. This allows for conditional questions, as follows:

```
Future<Boolean> q1Result = q1.invoke();
q2.invoke();
if(!q1Result.get())
{
    q3.invoke(); // Only invoke Q3 if the user got Q1 wrong.
}
q4.invoke();
```

(Note: the use of futures here would possibly be unnecessary, except for the next part.)

3.3 Next Question Preview

At the same time that the user is being shown one question, the *next* question (just the text, not the input widgets) must be shown either off-to-the-side or at the bottom of the screen.

This implies that invoking a question via the `invoke()` method (or equivalent) must actually be asynchronous. Hence the `Future` object; otherwise, the quiz plugin wouldn't get a chance to say what the next question actually is.

There is a possibility that the next question may not be available, either because this is the last question, or because the quiz plugin is waiting on the result of the current one. In this case, the preview can be left blank.

Hint! This might be a job for the `BlockingQueue.poll()` method (not that this is necessarily the only solution).

3.4 Restarting / Reloading

The application must provide an option (without simply exiting) to restart the quiz, or load a new quiz, while the current quiz is part-way through.

4 Constraints and Non-Functional Requirements

Your application must conform to several other requirements:

(a) You must use Gradle as the build tool. Each plugin must be a separate subproject, and the build dependencies must reflect what each plugin actually depends on.

(b) Related to the above, the plugins must be separately-distributable from each other and the main application. For instance, it must be possible to distribute the application to a potential user *without any* plugins, and then to distribute each plugin separately later on.

Though the application can't do anything without at least one quiz plugin and one question-type plugin, it must not require any specific plugins in order to run.

(c) It *may* (but does not necessarily need to) use any of the Apache Commons and/or Google Guava libraries. This will of course have implications for your build setup. You may not use any other third-party libraries.

5 Code and Design Quality

You must adhere to good SE practices, including (but not necessarily limited to):

(a) High cohesion and low coupling (separation of concerns). Don't bundle together diverse responsibilities in a single file, or share individual responsibilities across multiple files. (You should treat any anonymous/local/nested classes as *part* of their respective containing classes for this purpose.)

(b) Thorough commenting. Every top-level class and top-level method should have an attached comment. (You do not have to explicitly comment anonymous classes, but any relevant remarks should appear in the enclosing method's comment.)

(c) Proper error handling. Handle errors in the right place (which is not necessarily where the program first discovers them), and handle them sensibly.

(d) In-code referencing, if relevant. If you use external sources in preparing your code, you must cite them *in the code* at the relevant point(s). However, be aware that if you use external sources to short-circuit the design work involved in this assignment, *you may compromise your mark*.

Don't worry about citing material from the SEC lectures or worksheets, or from the standard Java API.

6 Plugin Writer's Manual

Prepare a set of documentation that explains how to write both (a) a quiz plugin, and (b) a question-type plugin, for your application. You may assume the reader of this document is a competent Java/Swing/JavaFX programmer, and is familiar with this

assignment specification. However, you must cover all other details required to build a plugin.

To approach this task, put yourself in the shoes of a plugin writer, not knowing anything about the design of the application. What level of documentation would you need to see?

Make use of example code where appropriate, though note that you should *explain* any examples you give, not just assume the reader will understand them on their own.

The Plugin Writer's Manual does not need to be long, as long as you cover all important points. You will want to ensure that your coding work is done before you write the manual.

7 Debugging Advice

The advice below does not add any extra requirements, but it may save you time!

- Use visualvm/jvisualvm, or any other reasonable profiler tool, to peek inside your application at runtime. You will be able to see what each of your threads is doing.
- Whenever your application performs some significant task, log it. You're almost certainly familiar with the practice of putting `println`s throughout your code to track down bugs. Logging is like this, but more systematic and pre-emptive.

  You can simply use `println`, but you may like to try out Java's real logging API. Remember to make your messages meaningful, so you can understand what's happening when you see them later on.

  Log messages are not part of the UI, so you can print them out to the console (and/or to a file).
- Whenever you catch an exception, whatever else you do, log it. However, be mindful that logging is *not* the same as telling the user that something went wrong (because it is not part of the UI).

8 Submission

Submit the following electronically to the Assignment 2 area on Blackboard:

- A completed Declaration of Originality;
- The project directory, including all build-related files and source code files in their appropriate subdirectories (but omitting any auto-generated files/subdirectories such as `build/` and `.gradle/`);
- The plugin writer's manual (in .pdf format).

You are responsible for ensuring that your submission is correct and not corrupted. If you make multiple submissions, only your newest submission will be marked. The late submission policy (see the Unit Outline) will be strictly enforced.

Please DO NOT use WinRAR!

9 Demonstration

You will be required to demonstrate and discuss your application with a marker in-person. This may include, for instance:

- Rebuilding and running your application.
- Creating and/or modifying plugin code as requested by the marker.
- Answering questions about any aspect of your submission.

You must either be available during the practical sessions in week 14, or schedule an alternate time (in agreement with the marker/lecturer) *before* the start of the exam weeks.

10 Academic Integrity

This is an assessable task. If you use someone else's work, or obtain someone else's assistance, to help complete part of the assignment that is intended for you to complete yourself, you will have compromised the assessment. You will not receive marks for any parts of your submission that are not your own original work.

Further, if you do not *reference* any external sources that you use, you are committing plagiarism and/or collusion, and penalties for Academic Misconduct may apply.

Please see Curtin's Academic Integrity website for information on academic misconduct (which includes plagiarism and collusion).

The unit coordinator may require you to provide an oral justification of, or to answer questions about, any piece of written work submitted in this unit. Your response(s) may be referred to as evidence in an Academic Misconduct inquiry.

End of Assignment