NATIONAL UNIVERSITY OF SINGAPORE

CS2103/T – SOFTWARE ENGINEERING

(Semester 1: AY2015/2016)

Time Allowed: 2 Hours

INSTRUCTIONS TO CANDIDATES

1. Please write your Student Number only. Do not write your name.
2. This assessment paper contains **SIX** questions and comprises **TEN** printed pages.
3. Answer **ALL** questions within the space in this booklet.
4. This is an OPEN BOOK assessment.

STUDENT NO: _____

This portion is for examiner's use only

Question	Marks	Remarks
Q1	/6	
Q2	/6	
Q3	/5	
Q4	/10	
Q5	/6	
Q6	/7	
Total	/40	

[Examiner comments]

- Do not take example answers given here as 'the only right answer'. In most cases it is possible to get full marks even if your answer differs from the example given.
- The project used in this paper is slightly harder and slightly different than the module project.
- This assessment paper does not cover project management. That aspect was covered in CA.

The **problem description** used for all questions in this assessment paper is given in the appendices (last two pages of this assessment paper).

Q1 [2+2+2=6 marks]:
Please read *Appendix A* of the problem description. When answering this question, you may consider the requirements specified in the problem description as well as requirements that you would like to add.

a) Assume that you have categorized CollatePlus user stories into categories *must-have*, *nice-to-have*, and *unlikely-to-have*. Give one *must-have* and two *nice-to-have* user stories for CollatePlus. All three user stories must help the user evaluate the code quality of the code written by a student.

Areas tested by this question: Functional requirements, product design
Sample answer:

1. Must have: As the lecturer, I can list all code written by a specific student so that I can examine the code written by that student.
2. Nice to have: As the lecturer, I can generate quality related statistics of code written by a student so that I can use those in evaluating the code quality of the student.
3. Nice to have: As the lecturer, I can view code with syntax coloring so that it is easier for me to understand the code.

Examiner comments:

- The must have user story must be something the user cannot do without. i.e. without it, the app will be useless. Giving such features as nice-to-have will be penalized too.
- The user stories should be about evaluating code quality, not other things such as work distribution, consistency of commits
- Some answers were out of scope. E.g. 'show test coverage of code written by a student'. Measuring test coverage requires running the code using a coverage tool, which is unlikely to be in the scope of a CollatePlus.
- The app is not meant for students. The user stories should not be for students.
- Answers that are not written in user story format will be penalized.

b) Give an example of a *non-functional requirement* of CollatePlus that is directly related to a functional requirement specified in your answer to part (a) above.

Areas tested by this question: non-functional requirements.
Sample answer:

- CollatePlus must be able to list up to 10 KLoC of lines written by a single student.

Examiner comments:

- Some answers were too vague (e.g. the software should be *reasonably fast*). Good requirements should be concrete and precise.
- Some were assumptions rather than requirements. E.g. The computer will have enough memory for the app. A better requirement is 'The app should need no more than 100Mb of

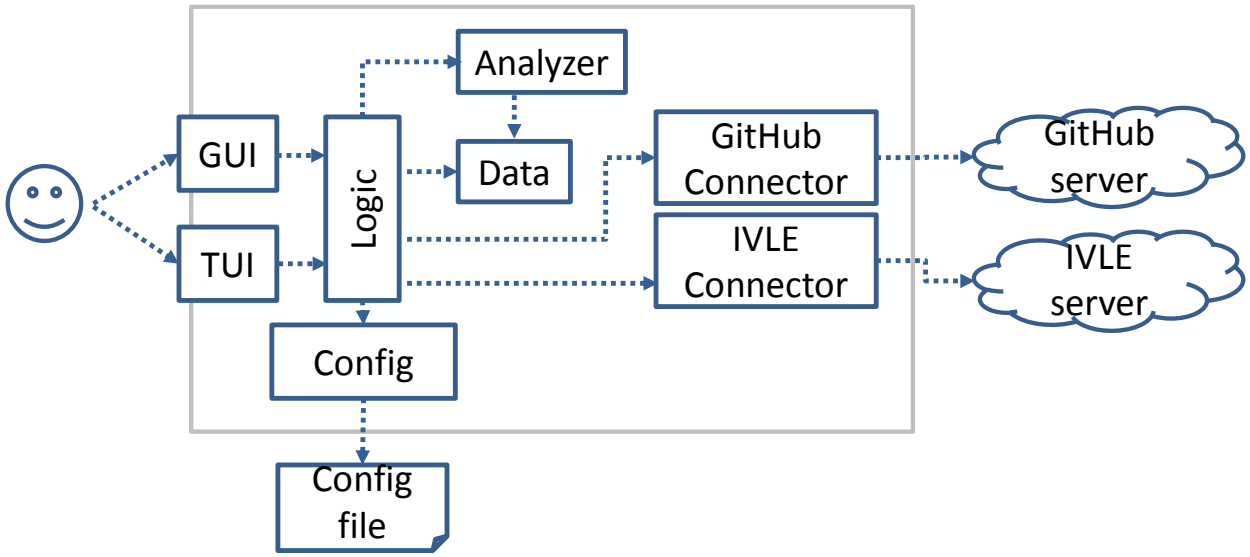
RAM'.

c) Give two *extensions* to the use case description given in *Appendix B*. Write your answer below (not in *Appendix B*).

Areas tested by this question: use cases.
Sample answer:
3a. App cannot connect to IVLE server or GitHub server
    3a1. App informs user that it cannot connect to server.
    Use case resumes at step 2. (also acceptable: Use case ends)
7a. The student does not exist
    7a1. App informs user that student does not exist.
    Use case resumes at step 7.
Examiner comments:
- Other possible extensions: use enters wrong command, configuration file is missing, ...

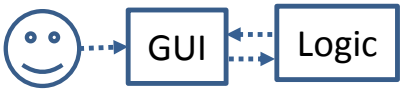
Q2 [4+2=6 marks]:
(a) Propose an architecture for CollatePlus. Your architecture must include the components given in *Appendix C*. If your architecture contains more components (recommended), please provide a brief overview of each component you added (similar to the descriptions given in *Appendix C*).

Areas tested: Architecture
Sample answer:
TUI: The Text UI
IVLE Connector: Handles connection to IVLE
GitHub Connector: Handles the connection to GitHub
Config: Handles reading of config values from hard disk.



Examiner comments:
- Other possible components: Parser, Util/Common, TestDriver
- Both solid arrows or dashed arrows are acceptable if used consistently. However, indiscriminate use of double-headed arrows (or pairs of arrows in two directions) will be penalized. If you use double-headed arrows everywhere, the reader cannot identify the direction of the dependency.

(b) Consider this extract (only two components are shown) from another proposed architecture for CollatePlus. Comment on how



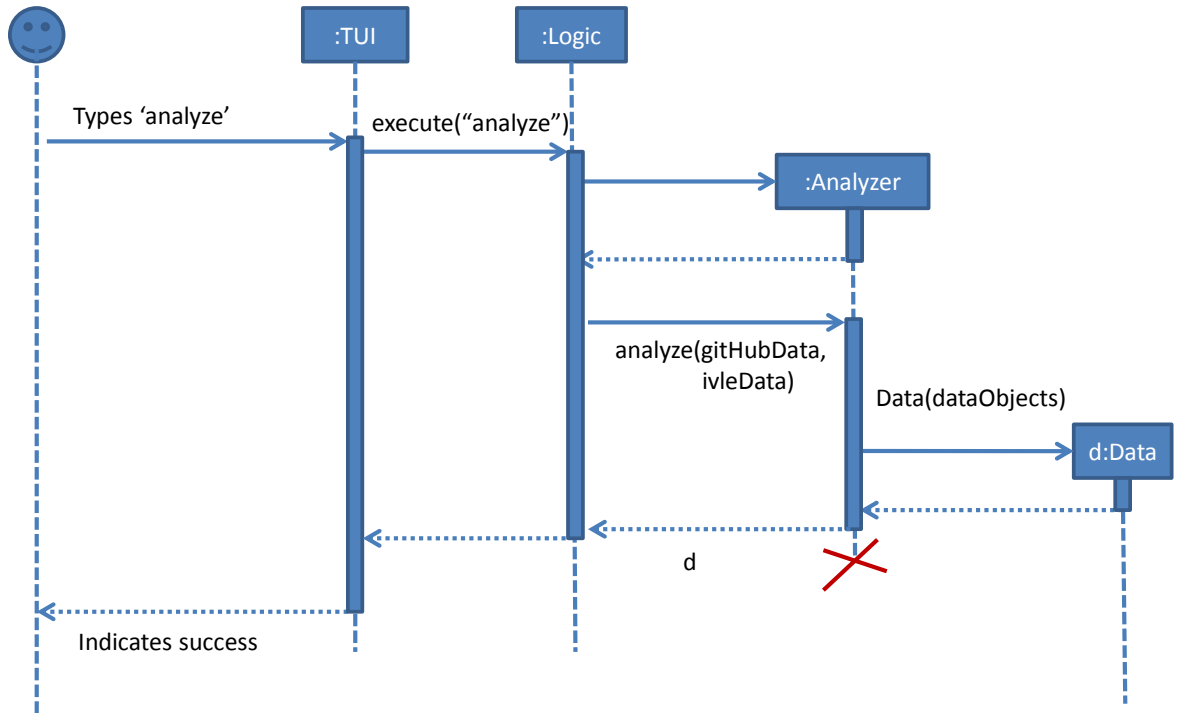
the two arrows between GUI and Logic affect the design quality.
Is it possible to remove one of them?
If yes, which one and how? If no, why?

Areas tested: Design patterns, design quality
Sample answer:
The two arrows indicate a high coupling between the two components. One can argue that Logic will be more cohesive if it did not depend on the GUI. Either of the two dependencies can be removed using the Observer pattern. Another simpler way to remove the dependency from Logic to GUI is to ensure that any change to GUI is initiated in GUI itself rather than by Logic. This is especially suitable here because the data of CollatePlus do not change over time (they are downloaded once at the beginning) and using Observer pattern in such a case would be an overkill.

Examiner comments:
- You are expected to refer to coupling and/or cohesion in the answer. Mentioning Observer pattern wasn't required but can earn bonus marks.

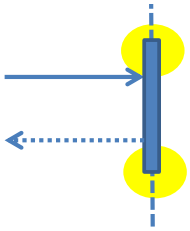
Q3[5 marks] Consider the use case description given in *Appendix B*.
Use a suitable UML diagram to explain the interactions between architectural components required during step 5-6 only.
State (in the diagram) the API methods taking part in the interactions.

Areas tested by this question: API design, Sequence diagrams
Sample answer:



Examiner comments:
- You are expected to show creation of Analyzer component and Data component.
- Bonus marks for showing the discarding of Analyzer component.

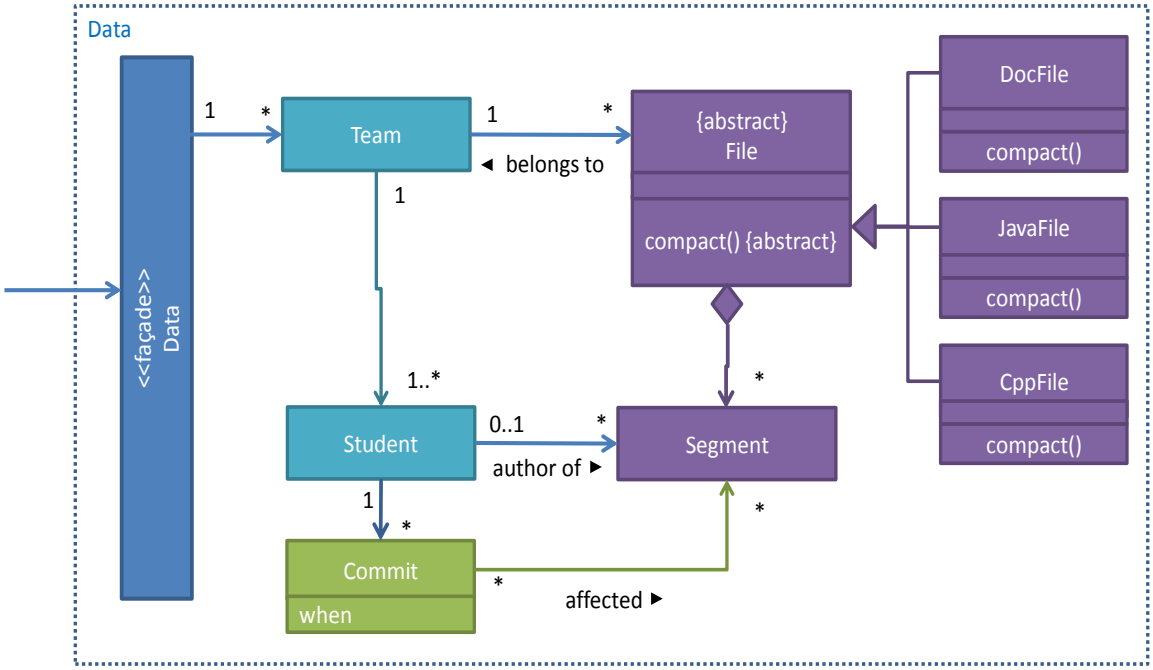
- Misusing activation bar (i.e. starts too early or continues beyond the return arrow) will be penalized.



Q4 [7+3=10 marks] (a) Use suitable UML diagrams to propose an object-oriented design for the `Data` component. Show <u>all</u> *navigabilities* and *multiplicities*. Try to minimize *associations*.
Show important *attributes* and important *methods* only (no more than 5 each).
Your design must include, but is not limited to, the classes listed in *Appendix D*.

Areas tested by this question: OOP, class diagrams

Sample answer:

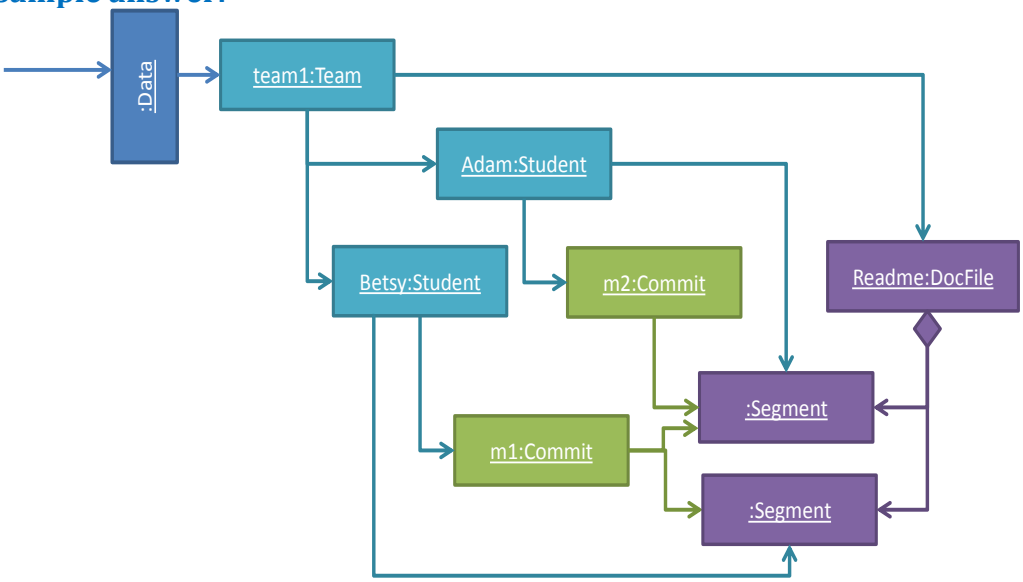


Examiner comments:

- Other possible classes (not strictly necessary): User, TeachingTeamMember, Repo
- Bonus marks for using inheritance, composition.
- Navigabilities may differ but all objects should be reachable.
- Note the Some Segments may not be claimed by anyone, hence the multiplicity of 0..1 in the Student—Segment association
- Each file belongs to a Team, not a Student, even if a single team wrote the entire file.
- It's ok to have a few more associations than given in the sample answer, but not too many, as the question asks you to minimize associations.

(b) Consider the sample data given in *Appendix E* and your proposed design above. Use a suitable UML diagram to show the object structure in the `Data` component for the given sample data.

Areas tested by this question: object diagrams
Sample answer:



Q5 [4+2=6 marks]:
(a) Consider the getStudents method/function in *Appendix F*. Use Java, C++, or pseudo code to show how you would use i. *assertions*, ii. *exceptions*, and iii. *logging* in that method.
Show no more than one example of each. Only the relevant lines of the method need to be shown.

Areas tested by this question: defensive coding
Sample answer:

```
assert (moduleCode != null);
...
try{
  //get student list
} catch (ServerNotConnectedException e){
 log("Server not connected when checking for module: " + moduleCode);
 throw e;
}
```

Examiner comments:

- Should not use assertions to confirm the server is up because although it is assumed the server connection has been verified before, the server can go down any time. Exceptions should be used instead. This was discussed during a lecture, taking Google servers as an example.
- Printing the stack trace or an error message to the console is not an acceptable ways of handling an exception. In fact this method should not print anything because we don't even know which UI (i.e. GUI or TUI) is being used. If this method interacts with the user, it breaks the principle of separation of concerns.
- Blatant coding standard violations might cost you marks. A common mistake was to have a line break between "}" and " "
- Avoid picking examples that are not clear cut, such as given below. As you are given the freedom to pick your own examples, you will not be given the 'benefit of the doubt'. If the example is not clear cut, you will not get marks.
    - assert numberOfStudents > 0 is not a good example of using asserts. Isn't it possible

that the module does not have students yet (e.g. at the beginning of the semester).
    - moduleCode == null is unlikely to be situation for throwing an exception. This more likely to be a programmer error rather than a user error or a problem in the environment.
    - Catching NullPointerException is not a good example either. This exception can happen anywhere when operations are performed on objects that are not confirmed as not null. That means it is a result of sloppy coding rather than user/environment errors.

(b) If the Analyzer component consists of just one class and it is instantiated no more than once at any time, is it OK to make all its methods and attributes class-level (i.e. static)? Justify your answer.

Areas tested by this question: advanced implementation techniques.
Sample answer:
Not recommended because making everything static will make it harder to test (because dependency injection is hard to use on static methods because static methods cannot be overridden) and it goes against the OOP philosophy.

Examiner comment:
- This was discussed during the lecture and in the IVLE forum. Nevertheless, this was intended to be a 'difficult' question and as expected, only about 20% answered it correctly.
- The question does not ask whether it is necessary to make everything static. If that was the question, the answer is a clear 'No'. It asks if it is OK to make everything static.
- Other weaker, but not totally wrong reasons for not making the class static:
    - It cannot be created/discarded at will, which means it will take up memory as long as the app is running. The reason why this is not a strong argument because we can have a method in the class (e.g. reset()) to set its variables to null, freeing up most of the memory held by it.
    - If multiple instances are needed in the future, we need to change the code a lot.
    - The state of the Analyzer becomes shared/global across the whole application which makes it more prone to misuse and a cause of subtle bugs.

Q6 [5+2=7 marks] (a) Assume you have been asked to test the getStudents method described in *Appendix F*. Propose an effective and efficient set of tests cases. You may not propose more than 8 test cases.
Given below is an example test case list used for testing a different method isCorrect(String answer) from a different software. You may follow a similar format in your answer.

#	answer	Is question open for submission?
1	Correct answer	Yes
2	Incorrect answer	No

Areas tested by this question: test case design heuristics

Sample answer:

#	moduleCode	User logged in?	A teacher of the module?	Server available?
1	valid	Yes	No	Yes
2	valid	No	Yes	Yes
3	valid	Yes	Yes	No
4	Non existent	Yes	Yes	Yes
5	null	Yes	Yes	Yes
6	Empty string	Yes	Yes	Yes
7	valid	Yes	Yes	Yes

Other parameter that can be considered: Is user an instructor?

Examiner comments:

- The key is to realize that the parameters of the function are not the only things that can be varied in test cases. Any other thing that can affect the behavior of a method should also be considered as inputs. The example given in the question was intended to remind you of this (note how it shows two inputs while the method being tested has only one parameter)
- To get full marks, you should have given at least three inputs (the sample answer above has four).
- You can also lose marks for not following other test case design heuristics. E.g. if you included multiple invalid values in the same test case.

(b) Comment on the following statement.

While 100% *path coverage* is very difficult to achieve, it is worthy of achieving because it certifies the code as bug free.

Areas tested by this question: coverage

Sample answer:

100% path coverage does not certify the code is bug free. Some needed paths may be missing from the code altogether. E.g. if the code has an if branch, but the else branch is missing altogether. In addition, 100% path coverage can be very expensive to achieve for non-trivial code.

Examiner comments: This was discussed during a lecture. Unfortunately, many did not answer this question correctly.

Given below is the **problem description** used for all questions in this assessment paper. You may detach this page from the assessment paper. There is no need to submit this page.

Appendix A. CollatePlus product description

Your team has been asked to build a software application called **CollatePlus** which will be used for analyzing text (code and documentation) written by students in CS2103 module project.

1. CollatePlus is a desktop application. It is meant to be used by CS2103 teaching team members.
2. CollatePlus can be accessed using a GUI or a Textual UI (TUI).
3. CollatePlus downloads student data from the IVLE server and commit history from the GitHub server. All data for the entire cohort is downloaded at once. The download is triggered by the user. Data from GitHub comes as a single string. Data from IVLE too comes as a single string. The data is then analyzed and converted into an object structure that captures the information required to display data about student contributions to the project.
4. For any student in the class, CollatePlus can show,
    a. A collation of all the text segments (code and documentation) written by the student for the project.
    b. A list of all the commits made by the student in the team repo.
    c. A list of all the project files modified by the student.
5. CollatePlus does not store the downloaded data in the user's Computer.
6. While multiple students may have modified a text segment, only one student may claim the authorship of a text segment. The author of a text segment is indicated using specially-formatted comments, similar to how you did it in your module project.
   Some segments may not be claimed by any student (e.g. code reused from elsewhere).
7. Some configuration data (e.g. login credentials of the user for IVLE/GitHub, mapping from matric number to github user ID, etc.) are to be put in a file named config.txt. The user is expected to create that file manually, following a specific format. CollatePlus reads that file at startup.

Appendix B. An example use case description

Use case: View text written by a student
Actors: Lecturer

1. Lecturer launches the Text UI version of the app.
2. App prompts the Lecturer to enter a command.
3. Lecturer enters the command to download data from servers.
4. CollatePlus downloads data from GitHub and IVLE.
5. Lecturer enters the command to analyze data.
6. CollatePlus analyzes the downloaded data and prepares the internal object structure.
7. Lecturer enters the command to view text written by a specific student.
8. CollatePlus shows a listing of all text written by the specified student.

Use case ends.

Appendix C. Suggested components for the architecture

- **GUI**: The Graphical User Interface.
- **Logic**: The main logic of the application.
- **Analyzer**: A temporary component used only during analysis of the downloaded data. The component is created at the beginning of analysis and discarded after the analysis.
- **Data**: Created by the Analyzer component. Holds the object structure containing the data required to answer user queries.

Appendix D. Suggested classes for the Data component

- Segment: An object of this class represents a segment of contiguous text authored by a particular student. E.g. if a student is claiming authorship of all text in a particular file, all text in that file is considered one Segment.
- File: Represents a file in a particular team's repo. This class supports a compact() method that removes unnecessary spaces from the file. What is considered 'unnecessary spaces' depends on whether the file is a documentation file, java file, c++ file, xml file, etc.
- Commit: Represents a commit made by a student. (note: CollatePlus only keeps track of who made the commit, when the commit was made, and which text segments were affected)

Appendix E. Sample data

- Adam and Betsy are in Team1.
- Adam has made only one commit. In that commit he added the file readme.txt with some content in it.
- Betsy has made only one commit. In that commit she added some text to the end of readme.txt and modified some of the Adam's text in it.
- Adam is claiming authorship of text segment he added. Betsy is claiming authorship of the segment she added.

Appendix F. getStudents method

getStudents (String moduleCode)

Connects to the IVLE server and retrieves a string containing details of students in the specified module if the logged in user is a teaching team member of that module. It is assumed that the connection to the IVLE has been set up before calling this method.

-- End of problem description--