NATIONAL UNIVERSITY OF SINGAPORE

CS2103/T – SOFTWARE ENGINEERING

(Semester 1 AY2014/2015)

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 **NINE** 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

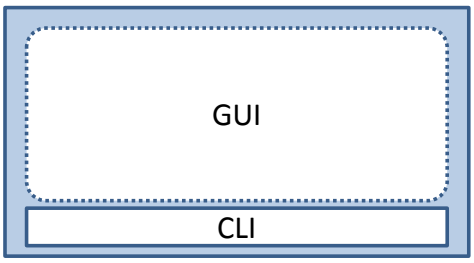
Question	Marks	Remarks
Q1	/5	
Q2	/5	
Q3	/10	
Q4	/10	
Q5	/5	
Q6	/5	
Total	/40	

Given below is the scenario used for all questions in this assessment paper:

The stuff in blue are comments inserted after grading student answers.

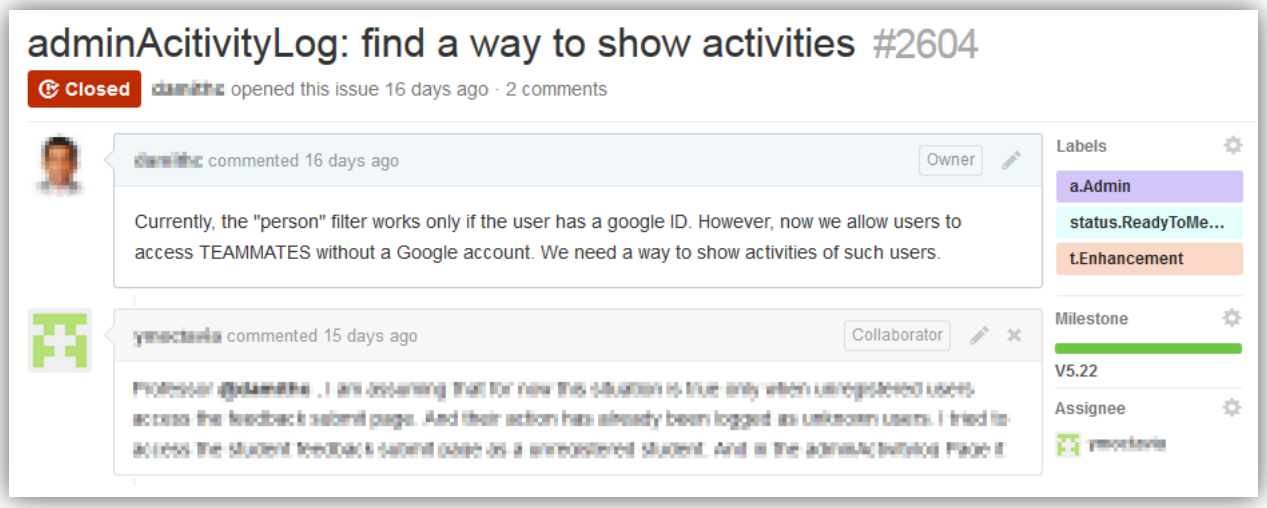
The project used in this paper is slightly harder and slightly different than the project you did in the module project.

Your team has been asked to build a desktop client for GitHub issue tracker. The name of the application is **HubIssues**. It is expected to provide a more convenient way to access the GitHub issue tracker so that users can work with GitHub issues without visiting the GitHub Website. Here is the expected behavior of HubIssues.



1. HubIssues has a GUI and a Command Line Interface (CLI). Most actions can be performed using either the GUI or the CLI, whichever the user finds more convenient.
2. HubIssues does not store issues locally (i.e. in the user's Computer). Instead, it connects to the GitHub remote APIs to read/write issues in the GitHub servers. However, user preferences are saved locally in a text file.
3. HubIssues synchronizes with issues on GitHub after each write operation on HubIssues. E.g. after a user edits an issue using HubIssues. In addition, users can force HubIssues to synchronize either by typing a command in the CLI or clicking a button on the GUI. There is no need to use multi-threading to auto-synchronize periodically.
4. HubIssues will show issues of only one project at a time.
5. HubIssues supports the concept of *issue hierarchies*. i.e users can designate an issue as the parent of another issue. For example, an issue representing an epic can be designated as the parent of issues representing the user stories related to that epic. You can assume that the GitHub API supports issue hierarchies although the current GitHub Web interface does not.
6. HubIssues ignores issues that represent *pull requests*. If you do not know what pull requests are, you can safely ignore this point.
7. HubIssues does not interact with either the local Git repo or the remote Git repo on GitHub.

Given below is a screenshot of an issue in the GitHub issue tracker (Web UI), to refresh your memory.



Note: All questions in this assessment paper are interlinked. Please answer sequentially. Informally-drawn UML sketches with just enough information are acceptable. Answers may be written using a pencil.

Q1 [1+2+2 = 5 marks]:
Assume you have categorized HubIssues user stories into categories *must-have*, *nice-to-have*, and *unlikely-to-have*.
When answering this question, you can consider the requirements specified in the previous page as well as requirements you would like to add yourself.

This question covers Requirements, Product design
At least one of the user stories given for (a) or (b) should be in full format, including the benefit. If not, I have no way to know if you know how to write a proper user story.

a) Give one *must-have* user story for HubIssues.
e.g. As a user, I can create an issue so that I can add more issues to the issue list.

b) Give one *nice-to-have* user story for HubIssues. It should be a user story that makes HubIssues more useful than the GitHub Web interface.
e.g. As a user, I can view issues of different milestones side-by-side so that I can do milestone planning more easily than with the Web interface.

c) Give one non-functional requirement for HubIssues. It should be a requirement that is specific to HubIssues (as opposed to a requirement that is applicable to many other software).
e.g. The use of HubIssues by some project members should not affect other project members who are not using HubIssues.

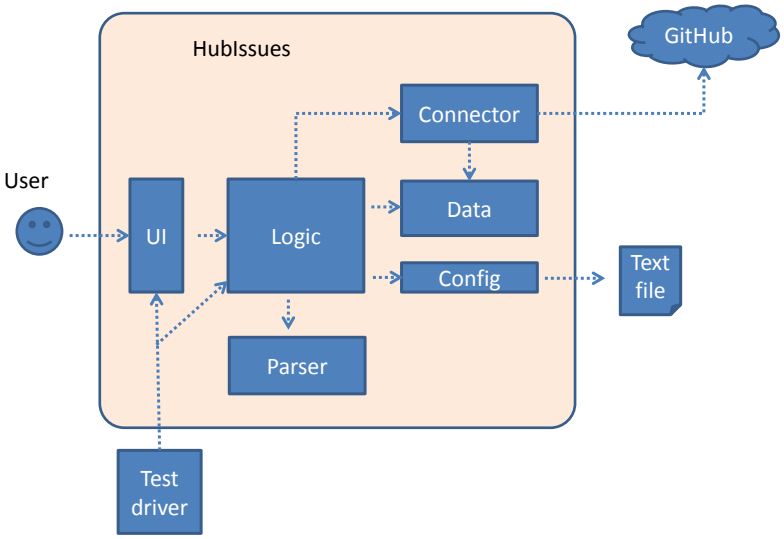
Many gave wrong answers to this question because they did not know the correct definition of *non-functional requirements*. They are defined in the handouts. Note that non-functional requirements are not given as user stories. User stories are *functional requirements*.

Q2 [5 marks]:

Propose an architecture for HubIssues that includes the following high-level components. You may not leave out any of the given components, but you may add more components. You are expected to minimize coupling and maximize cohesion in your architecture.

- `Parser` : Parses the CLI commands typed by the user
- `Data` : The in-memory container for objects representing HubIssues data. E.g. Issues, Users, Labels, Milestones, Comments, etc.
- `Connector`: Handles the connection with the GitHub remote API.

This question covers: Architecture, Coupling, Cohesion
Here is an example.



To get full marks, you should have most of the below:

- Show external connections (e.g. which component talks to GitHub)
- Use appropriate symbols e.g. don't use boxes for everything
- Components laid out in a meaningful way so that it is easier to understand
- Arrows used meaningfully (no indiscriminate use of double headed arrows)
- Reasonable coupling and cohesion levels e.g. It is better for the Connector to depend on Data rather than the other way around (why should Data know anything about a connection to a Server?)

A general note: When grading, I'm trying to evaluate how good you are as a software engineer. Even if two diagrams contain the same content, I'm going to give higher marks to the diagram that is easier to understand because effective communication is a necessary skill for a software engineering.

Q3 [10 marks]:

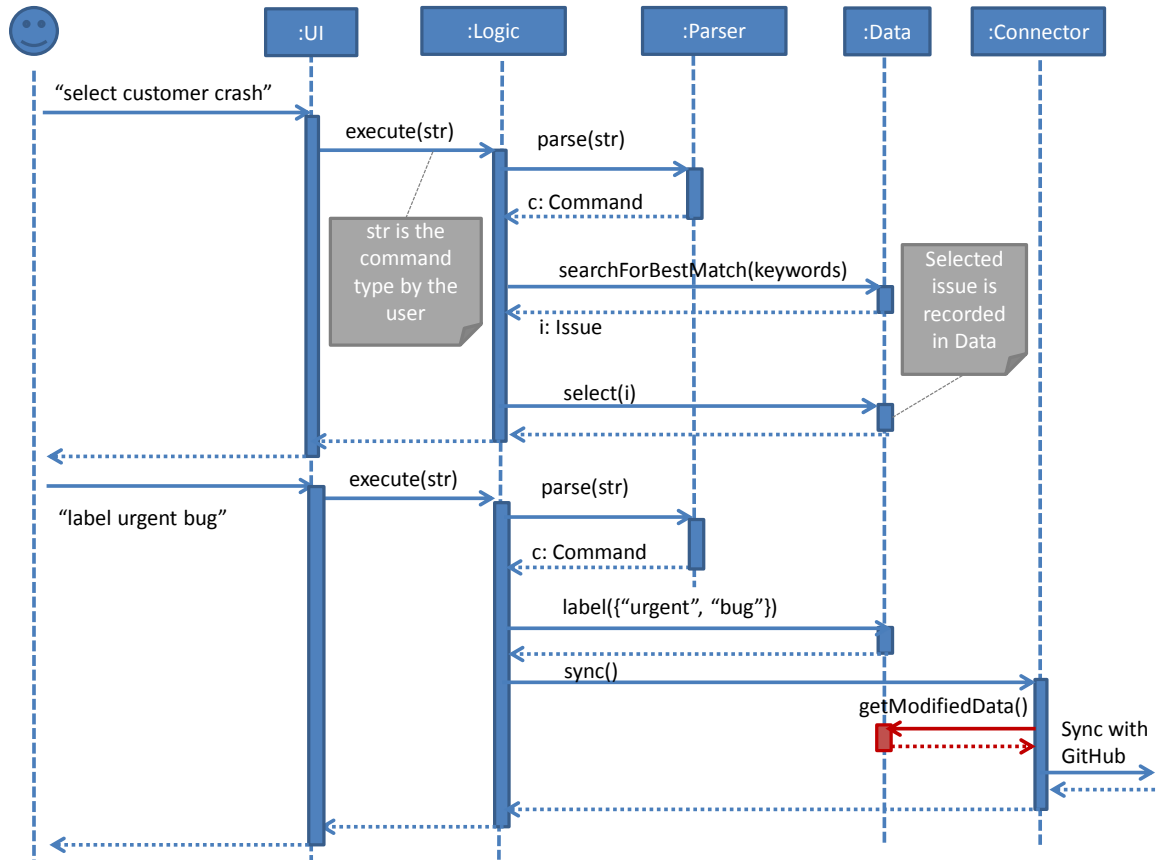
Given below are two CLI commands available in HubIssues.

`select` `keyword1 [keyword2]…` Selects the issue that is the best match for the keywords specified. e.g. **`select`** `customer crash minimize` The above example will search issue titles for words `customer`, `crash`, and `minimize` and choose the issue with the 'best matching' title as the 'selected' issue. This chosen issue will remain as the 'selected' issue until another issue is selected.
`label` `label1 [label2] …` Adds one or more labels to the currently 'selected' issue. e.g. **`label`** `urgent bug ui` The above example will add the labels `urgent`, `bug`, and `ui` to the currently selected issue.

Use suitable UML diagram(s) to explain the interactions between architectural components required to execute the command **`select`** `window view` followed by the command **`label`** `bug medium`
The diagram(s) should show the API methods used in the interactions.

Coverage: API design, Sequence diagrams

Here is a reasonable answer. Your answer may differ based on your architecture and other design decisions:



The most appropriate diagram for this situation is an SD. Those who chose other type of diagrams did not get many marks.

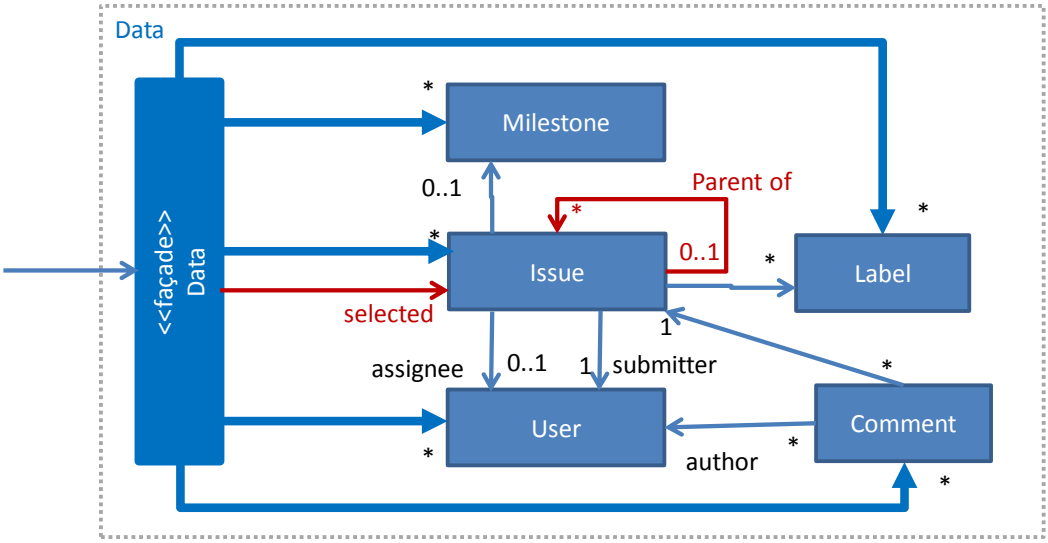
To get full marks:

- Use the correct sequence diagram notation e.g. colon before component name, no underline for component name, dashed line for return arrow, activation bar ends at return arrow
- Drawn at architecture level. Interactions between low level components are not shown.
- The interactions make sense. For example, it should be clear how the Connector component gets data from the Data component for syncing with GitHub, shown in red in the diagram above. Many answers contained only a simple sync() call which is not enough.
- Most of the methods called are shown formally i.e. show method name and parameters

Q4 [10 marks]:
Propose an object-oriented design for the `Data` component.
Include navigabilities, multiplicities, association roles/labels etc. when they add value to the diagram.

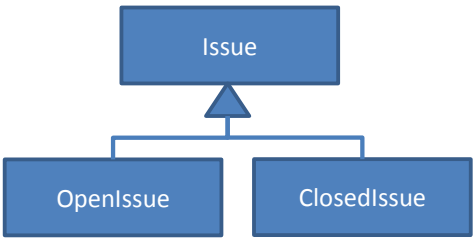
Coverage: Design, OOP, class diagrams
Expected diagram, a class diagram, possibly complemented with an object diagram.

Here is a reasonable answer.

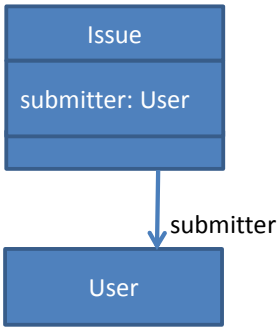


To get full marks:

- Enough OO applied. That is, most of the data (Issues, Labels, Milestones, User, Comment, etc.) should be classes. For example, a label cannot be a simple String because labels have colors.
- No wrong use of inheritance. For example, the following is not correct. Open and Closed issues are not different types of issues. They differ simply based on the status of the issue, which can be an attribute of an Issue.



- No wrong use of composition. For example, Milestones do not 'compose of' (i.e. filled diamond) of Issues. That implies that Issues are part of a milestone and when a milestone is deleted, all issues assigned to it are deleted, which is not true.
- No unnecessary classes: For example, Assignee, Creator, SelectedIssue, are associations, not classes. At best, they can be association classes only.
- Correct notation used: No mixing up of class diagram notation with object diagram notation (e.g. . underlined class names). Some used arrows with big heads which could either be inheritance or associations, confusing the reader.
- No confusion between association links and attributes: Some showed both association and the attribute for the same thing. Here is an example:



A general note: Average performance for this question was lower than what I hoped for. Most answers were not OO enough (had just one or two gigantic classes). Please continue to improve your OO thinking. OO favors small classes with single responsibilities.

Q5 [3+2=5 marks]:
(a) Propose *system test* cases for the **`label`** command described in Q3. One example is provided.
Design test cases in a way that maximize the efficiency and effectiveness of testing.
Note: Please read both part (a) and (b) before answering part (a) of this question.

Input	Rationale
`1. label bug ui`	Typical, correct use of the command.

Coverage: Testing
Here are some interesting test cases:

Input	Rationale
`label`	No labels specified. Error expected.
`label label`	Keyword used as a label. Should be accepted.
`label bug bug`	Repeated label
`label bug` `label bug`	Applying the same label again. Should be accepted.
`{no issue selected}` `label bug`	Applying label without selecting an issue first. Should result in an error.
`LaBel label1`	To check case sensitivity of the command
`label critical` `label CRITical`	Applying the same label again but using a different case. Should be accepted.
`label non-existent-label`	Applying a label that is not previously defined as a label

There are many other interesting test cases: using foreign language characters in the label, labels with spaces in the middle, having extra white spaces (e.g. " `label  bug  `"), using wrong separator (e.g. "`label bug,window`"), empty string as a label (e.g. "`label  `"), using escape characters in the label, etc.
However, passing `null` as a label is not a valid test case. This is a system test. There is no way the user can pass a null to the system by typing in the CLI.
Some of you gave test cases for *conflicting labels* such as 'priority.high' and 'priority.low'. This is not a relevant test case because neither GitHub nor HubIssues recognizes *exclusive label groups*. That feature was in another similar product called HubTurbo that I mentioned to you sometime back.
To get full marks, you need at least 4 interesting test cases, not counting uninteresting test cases. Unfortunately, most (~80%) did not have enough test cases to get full marks.
Please try to be more creative and thorough in testing. You cannot afford to be 'gentle' when testing. Your success as a software engineer largely depends on how good you are in testing, even if you don't take a testing job.

(b) Explain one example of how you used equivalence partitions and one example of how you used boundary values when designing test cases listed above.

- One of the simplest example of equivalence partitions is [existing labels] vs [non-existent labels]. These two partitions do not have boundaries.

Another example is [enough labels] vs [not enough labels]. The [enough labels] partition covers 'one or more labels' situation and therefore has the boundary of one label.

- Length of the command word `label` (i.e. 5) cannot be used for boundary value analysis. However, maximum length of a label (if there is such a limit specified) is suitable for boundary value analysis.

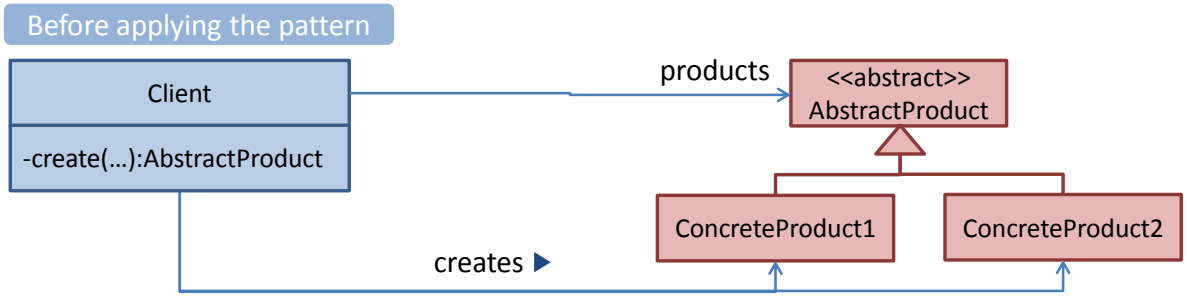
Q6 [5 marks]:

Assume you are already using the *Command pattern* in your HubIssues design. Would the *Simplified Factory pattern* (given below) apply in your design? Justify your answer.

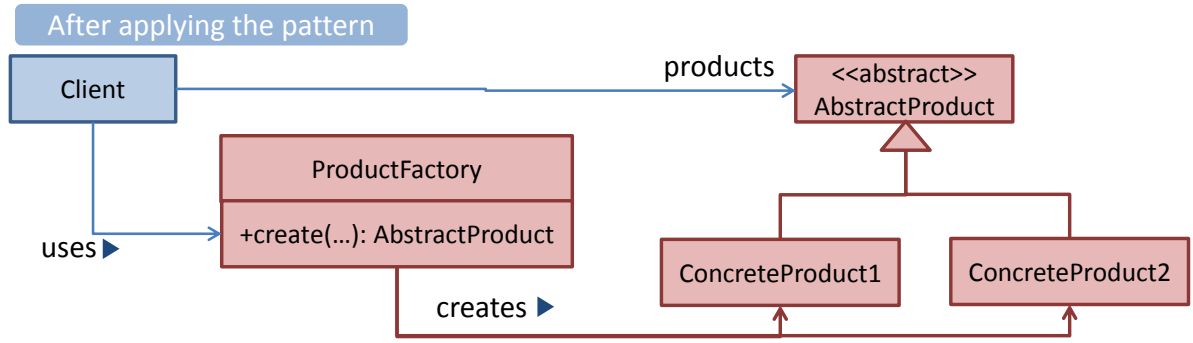
Pattern: Simplified Factory

Context: A class has to create objects of different subclasses. E.g. the `Client` class below needs to create `ConcreteProduct1` objects or `ConcreteProduct2` objects.

Problem: The logic for selecting which subclass to create complicates the class unnecessarily and makes the class unnecessarily dependent on the subclasses in concern.



Solution: Move the 'selection logic' to another class, which then provides a method to create objects.



Coverage: Design patterns, The ability to learn new patterns

If Command pattern is applied already, that means some part of the code needs to select which concrete command to create. Then it is a question of whether the Simplified Factory Pattern can be applied to isolate the logic that selects which command to create. If you were able to identify this link between the two patterns, you earned almost 4 marks. The remainder was for arguing if the SFP to be used or not. I favored the 'yes' answer, but gave marks for 'no' answer as well, as long as the argument was sensible. There was no right or wrong answer.

BTW, note that SFP was a made up pattern, based on an actual pattern called the Factory method pattern.

The performance for this question was better than I expected. That was a pleasant surprise for me ☺

--- End of Paper---