Given in the following sections are the qualities I looked for in your answers. The more of them you had in your answer, the higher your score could be. Note that you did not need to have all of them to score full marks.

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

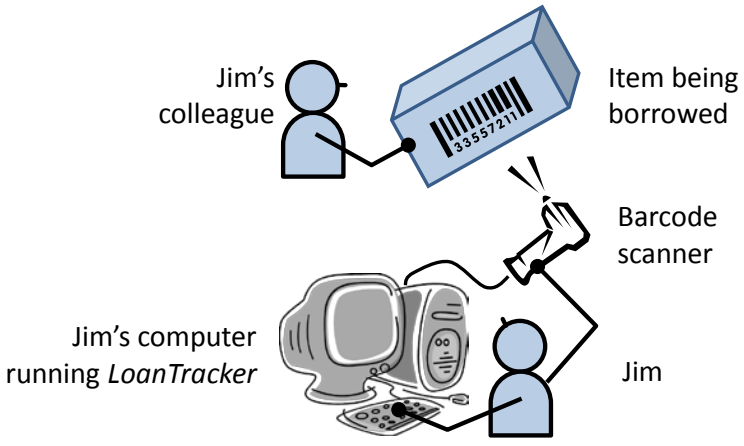
Jim (the target user of your module project) has requested your team to build another product for him. In his workplace, Jim is the caretaker of a collection of items that are available for employees to borrow for short periods (e.g. to use during business trips). Some frequently borrowed items in the collection are: 20 thumb drives, 12 iPads, 10 laser pointers, and 15 laptop carriers. He wants your team to build a '**LoanTracker**' system to keep track of loaning of these items. Currently, Jim maintains loan records in a spreadsheet, which has grown too big for his liking. He hopes the new system can help him keep track of past and current loans more easily. Furthermore, he hopes the new system will allow him to record requests by colleagues to reserve an item for a period in the future, a facility not currently available. In addition, you are free to add more features that may be useful to Jim.

The system is to be used by Jim only. It is not meant to be a multiuser system. Other employees still come to Jim for borrowing items.

The new system can use a barcode scanner that Jim has. The scanner can be connected to Jim's computer. The API provided by the scanner software allows you to retrieve the most recent *n* values the scanner has read where *n* is a parameter to be specified by the caller. The plan is to assign a unique number to each item in the collection, print those numbers as barcodes, and stick them on each of the items. Similarly, the barcode scanner can be used to scan employee ID cards to get the Employee ID of the borrower (a 12 digit number). Jim thinks the new system will allow him to record a loan by scanning the employee ID card and the item to be borrowed, followed by typing some more information if required (e.g. expected return date). Returning of items is expected to be recorded similarly.

Yes, Jim still likes typing more than using the mouse. As before, Jim wants data to be stored in a human editable text file.

Your team has six weeks (fulltime) to complete the project.



Q1 [8 marks]:
Propose an iterative delivery schedule for the project. Include three versions of the software delivered to Jim at various points of the project. Use *user stories* to specify the functionality delivered in each version. You may omit the *role* segment of the user story. You may also omit the *benefit* segment of the user story for basic features where the benefit is obvious.

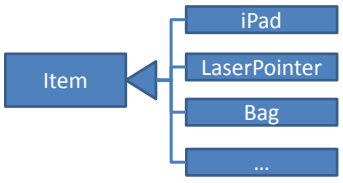
A good answer should have at least some of the following qualities:

- Uses user stories to specify deliverables (as required by the question), instead of other things such as feature lists or use cases.
- At least some examples of a properly written user story, including the 'benefit' (you were allowed to omit the benefit segment for basic features only)
- The proposed first version is 'complete' at some level (similar to V0.1 of the module project) so that Jim can try it out. Core features should be delivered early. Randomly dividing features among milestones is not acceptable.
- Enough time allocated for the first version. Given that the first version needs to be 'complete' at some level, it is unlikely you can deliver it very early.
- At least some user stories that are not immediately obvious from the description. E.g.
    - Upload data from a spreadsheet so that past data can be captured in the new system.
    - List details of 'in stock' units of an item type so that Jim can see if any more of that item type are left for loaning.
    - List return dates of items of a particular type so that Jim can know when more items of that type will be available for loaning.
    - List reservations of items of a particular type so that Jim can know if anybody is waiting to loan that item type.
    - Show the loan history of a person so that Jim can tell a colleague if he/she has any unreturned loans.
    - Show loan history of a particular item so that Jim can find who may have damaged a particular item while on loan.
    - Show overdue items so that Jim can request them to be returned.
    - Allow editing of scanned-in values, so that Jim can use the system even if the scanner is not working properly.
    - Undo an action so that Jim can correct mistakes in commands.
    - Use shorter commands so that Jim can perform actions faster.
    - Calculate utilization factor of an item type so that Jim can know which item types need increased stock levels.

Q2 [20 marks]:
Propose a design for the product. In your design, try to keep the *data* classes decoupled from classes that deal with *data persistence* so that you can change the persistency strategy (from text file, to something else) in the future without affecting the data classes.
The target audience for this description is your former CS2103/T lecturer who will be evaluating the quality of your design based on this description.

A good answer should have at least some of the following qualities:

- The design description is top-down. It should start from the high level (e.g. architecture) and progressively drill down to finer details. All-in-one diagrams (e.g. one class diagram for the entire system) were penalized heavily, as warned during lectures.
- Diagrams are drawn at different levels. This is related to the above point. There should be high-level diagrams as well as lower-level diagrams (e.g. class diagrams of a component).
- Contains both structure diagrams and behavior diagrams.
- At least some API definitions. E.g. Some API methods visible in the a sequence diagram
- Data and persistency are decoupled. For example, there is a separate class/component dealing with persistency only.
- Sufficiently Object-oriented. At least some of these concepts should appear as classes: `Loan`, `Reservation`, `Employee`, `LoanItem`. Related to that, `Loan` and `Reservation` could be modelled as *association classes*. Having 'monster' classes that do too many things (low cohesion) was penalized.
- The description is well balanced. Focusing too much on one component is a symptom of not knowing about parts done by other team members in the project.
- Some evidence of applying design patterns. Some applicable patterns were: Abstractions Occurrence (`ItemType - ItemCopy`), Command pattern, Façade pattern
- Notations used in the diagrams are correct. Omitting unimportant adornments is OK, but incorrect use of notation was penalized.
- Enough diagrams, less text. Concise and legible text descriptions.
- Design flexible enough. For example, the following is a bad design (in fact, this is an anti-pattern mentioned under the abstraction occurrence pattern). It requires adding a new class whenever a new type of item is added to Jim's collection.



Another weakness was to limit the design to the types of items mentioned in the requirements (those were given as examples only) and using the given item counts as multiplicities. Sure, Jim has 12 iPads now, but he might have a different number next week. The design should not be fixed to 12 iPads ('*' is more suitable as the multiplicity).

Q3 [3x4=12 marks]:
Give one concrete examples each from your proposed implementation of the *Loan Tracker* system for (a) – (d).
(a) An example of <u>not</u> following the *Liscov Substitution Principle (LSP)*.

A good answer should illustrate how a child class imposing a stricter constraint can make it 'un-substitutable', thus breaking LSP.

(b) An example where the implementation can be less/more *defensive*.

A good answer should give an example of writing extra code to prevent non-typical/unintended undesirable situations. Writing such code will increase defensiveness and not writing such code will decrease the defensiveness.

(c) An example where *Boundary Value Analysis* is useful in improving the efficiency and effectiveness of testing.

A good answer should identify one or more boundary values, the equivalence partitions of which those values are 'boundaries', and some indication of how this technique improves E&E of testing.

(d) An example where a feature passes *system testing* but fails *acceptance testing*, illustrating the difference between the two.

A good answer should give an example of a feature that meets the specification but does not fulfil the user need. Another acceptable answer is a feature that worked in the dev environment but did not work in the deployment environment.