

CZ2002 Object Oriented Design & Programming

Group Project

Lab Group XXX Group XXX

Prepared by:

Wu Wenxuan U1720945K

Tang Hoong Jing U1721417E

Tan Chye Hong, Jordan U1722016G

Yeo Lai Xiang, Leon UXXXXXXXX

Lance Gabriel UXXXXXXXX

Contents

[1. Declaration of Original Work for CZ2002 Assignment 3](#_Toc529992967)

[2. Executive Summary 4](#_Toc529992968)

[3. Design Considerations 5](#_Toc529992969)

[4. UML Class Diagram 6](#_Toc529992970)

[5. UML Sequence Diagram of “Print Student Transcript” 7](#_Toc529992971)

[6. Required Functionalities 8](#_Toc529992972)

[7. Test Cases and Results 9](#_Toc529992973)

## 1. Declaration of Original Work for CZ2002 Assignment

We hereby declare that the attached group assignment has been researched, undertaken, completed and  
 submitted as a collective effort by the group members listed below.

We have honored the principles of academic integrity and have upheld Student Code of Academic  
 Conduct in the completion of this work.

We understand that if plagiarism is found in the assignment, then lower marks or no marks will be  
 awarded for the assessed work. In addition, disciplinary actions may be taken.

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Course | Lab Group | Signature/ Date |
| Wu Wenxuan | BCG | SS4 |  |
| Tang Hoong Jing | BCG | SS4 |  |
| Tan Chye Hong, Jordan | BCG | SS4 |  |
| Yeo Lai Xiang, Leon | BCG | SS4 |  |
| Lance Gabriel | CS | SS4 |  |

## 2. Design Considerations

With the Single Responsibility Principle (SRP) in mind, our group has defined our classes to include attributes and methods that are directly related to the classes themselves. In doing so, we ensure that each class only assumes a single responsibility and there will not be more than one reason for any class to change. Furthermore, we noticed that some of our classes contribute to realizing the same underlying requirement. For example, both lecture and tutorial classes deal with course components. Consequently, we have opted to group such classes under the same package for organization and communication purposes.

Except for the *SCRAME* class, every class deals with changes pertaining to their own class and can therefore be represented as ‘entities’. The *SCRAME* class however, acts as a ‘control’ as it is responsible for calling the methods of other classes to coordinate and realize use cases. It also acts as the ‘boundary’ since it is through this class that the user interacts with the system.

At the highest level of abstraction, all classes implement the ‘serializable’ interface. After serialization, the entire state of the application is saved into a binary file and can be deserialized during the next execution of the program. This allows users to avoid entering past data into the application every time the application is restarted.

Our design illustrates various relationships among entities, as described by the following examples. The Subcomponent and MainComponent concrete classes are a *realization* of the Examinable interface. This implementation is aligned with the Open-Closed Principle (OCP), where the Examinable class is open for extension (other types of components can be added in the future) but closed for modification (interface details need not be changed). Observe that at this stage, we have included both the Serializable and Examinable interfaces in our design. Having two different interfaces support the Interface Segregation Principle (ISP) since not every class is required to implement the Examinable interface, while every class has to implement the Serializable interface for the entire application to be saved.

The CourseComponent superclass is a *generalization* of the Laboratory, Lecture and Tutorial subclasses. This supports the Liskov Substitution Principle (LSP), as the user of the CourseComponent superclass can continue to function properly if either of its subclasses are passed to the user. This is due to the design by contract where the preconditions of the subclasses are no stronger than that of the superclass, and the post-conditions of the subclasses are no weaker than that of the superclass. Furthermore, this relationship demonstrates the Dependency Injection Principle (DIP) since the superclass need not depend on the details of its subclasses.

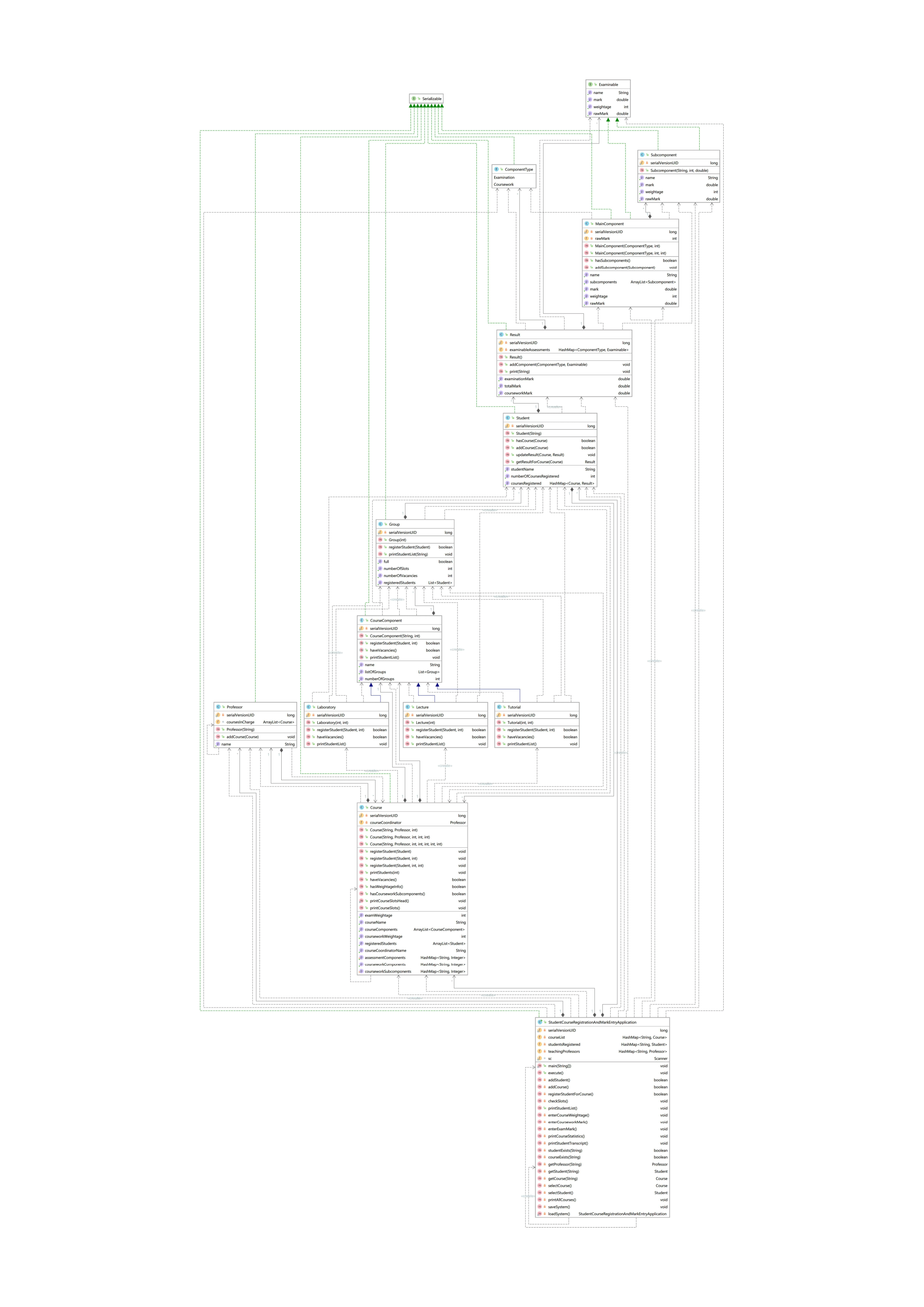
Each MainComponent is a *composition* of different Subcomponents. While the relationships described thus far can be classified as *association* relationships, we have *dependency* relationships as well. An example of this is from Student to Result, and this is due to the getResultForCourse() method in the Student class returning an object of Result type.

## 

## 3. Assumptions Made

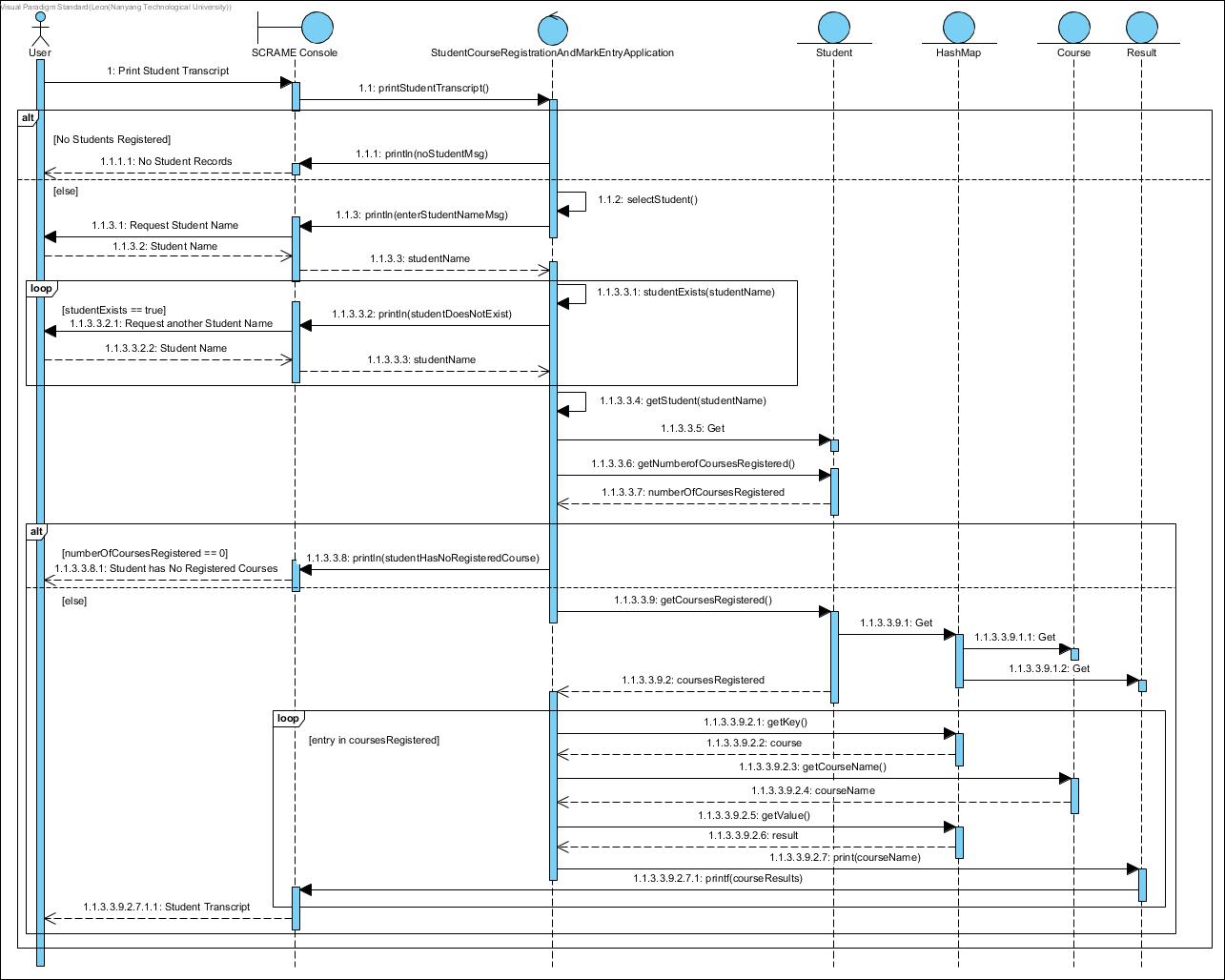
1. Each course will either only have lectures, or both lectures and tutorials or all lectures, tutorials and laboratories. This means that all courses must have lectures. And if a course does not have tutorials, it is impossible that it can have laboratories.
2. Each course has only 1 lecture group. When students register for a course, he/she will be automatically allocated to the only 1 lecture group.
3. Each course has only 1 professor/course coordinator.
4. There is no students/professor that has the same name.
5. There is no restriction to how many courses a student can take/a professor can teach.
6. A course will always have 2 main assessment components, namely Coursework and Examination.
7. The coursework component can have or not have subcomponents. If it has subcomponents, it must have at least 2 subcomponents.
8. Subcomponents cannot have more subcomponents on its own. That is,
9. All the weightages (as of parent component) of assessment components are integer values. For instance, 30%, 35% but not 35.5%. Although weightages can be double data type as an overall percentage. (e.g. 12.5%).
10. The coursework/examination marks are only entered once, that is at the end of the semester.

## 4. UML Class Diagram



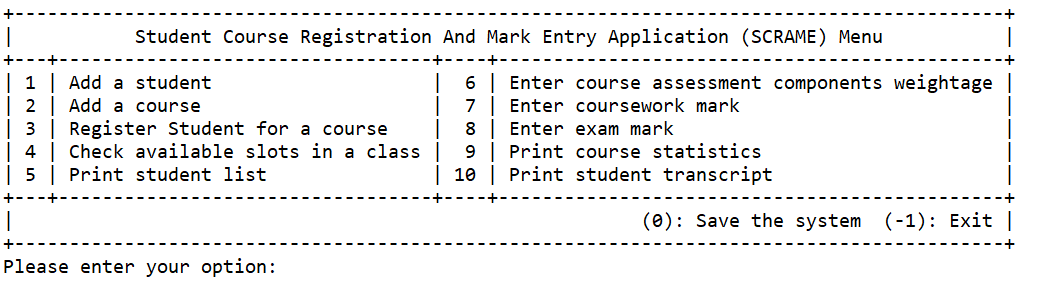
## 

## 5. UML Sequence Diagram of “Print Student Transcript



## 6. Test Cases and Results

**SRAME Menu options**



1. Add a student

|  |  |  |
| --- | --- | --- |
|  | **Test Case** | **Expected Outcome** |
| a | Add a new student |  |
| b | Add an existing student |  |
| c | Invalid data entries | To add in student name invalid entry:   1. Numbers 2. Name with number |
| d |  |  |

1. Add a course

|  |  |  |
| --- | --- | --- |
|  | **Test Case** | **Expected Outcome** |
| a | Add a new course  (with combination of (ii) from above) |  |
| b | Add an existing course |  |
| c | Invalid data entries | To add in prof name invalid entry:   1. Numbers 2. Name with number |
| d |  |  |

1. Register student for a course

|  |  |  |
| --- | --- | --- |
|  | **Test Case** | **Expected Outcome** |
| a | Add a student to a course with available vacancies in Tut/ Lab. |  |
| b | (i)Add a student to a course with 0 vacancies in lecture slot.  (ii) Add a student to a course with 0 available vacancies in Tut / Lab. | (i)  (ii) |
| c | Register the same course again. |  |
| d | Invalid data entries (eg wrong student ID / course code etc) | Wrong student name:  Wrong course code: |
| d |  |  |

1. Check available slot in a class (Vacancy in a class)

|  |  |  |
| --- | --- | --- |
|  | **Test Case** | **Expected Outcome** |
| a | Check for  (i) Tutorial class  (ii) Lab class |  |
| c | Invalid data entries(eg course code, class code etc) | Invalid Course Name: |
| d |  |  |

1. Print Student list by lecture, tutorial or laboratory session for a course.

|  |  |  |
| --- | --- | --- |
|  | **Test Case** | **Expected Outcome** |
| a | Print list by  (i) Lecture  (ii) Tutorial group/s  (iii) Lab group/s | **(i)**    **(ii)**    **(iii)**  Inserting image... |
| c | **Invalid data entries (eg course code, class code etc)** | **Invalid input for course name:**  **Invalid input for print student by selection:** |
| d |  |  |

1. Enter course assessment components weightage

|  |  |  |
| --- | --- | --- |
|  | **Test Case** | **Expected Outcome** |
| a | Enter course assessment with only exam + 1 main coursework component without sub-components (e.g. use the case stated in 4(e) in main section) |  |
| b | Enter course assessment with only exam + a main coursework component with 2 sub-components (eg use the case stated in 4(e) in main section) |  |
| c | Invalid data entries (eg course code, weightage percentage does not tally etc) | **Weightage doesn’t tally:** |
| d |  |  |

1. Enter coursework mark – inclusive of its components.

(Note: All mark entries should be based on 100 marks. Your application will eventually scale  
the marks to its component weightage percentage)

|  |  |  |
| --- | --- | --- |
|  | **Test Case** | **Expected Outcome** |
| a | Enter valid coursework mark for the course with only 1 main component |  |
| b | Enter valid coursework marks for course with 2 sub-components |  |
| c | Invalid data entries(eg course code, student ID, mark range etc) | Keying in course marks more than 100 or less than 0 |
| d |  |  |

1. Enter exam mark

(Note: All mark entries should be based on 100 marks. Your application will eventually scale  
the marks to its component weightage percentage)

|  |  |  |
| --- | --- | --- |
|  | **Test Case** | **Expected Outcome** |
| a | Enter valid exam mark for the valid course |  |
| c | Invalid data entries(eg course code, student ID, mark range etc) | Key in exam marks more than 100 or less than 0 |
| d |  |  |

1. Print Course statistics

(Note: You should have at least 15 students’ results for 1 course pre-populated)

|  |  |  |
| --- | --- | --- |
|  | **Test Case** | **Expected Outcome** |
| a | Enter valid course code |  |
| b | Invalid data entries(e.g course code, mark range etc) |  |
| c |  |  |

1. Print student transcript.

(Note: For this function, besides the individual overall course mark and grade, it will also print  
the student’s individual component marks – exam, coursework, subcomponents. The configured  
weightages should be displayed as well.)

|  |  |  |
| --- | --- | --- |
|  | **Test Case** | **Expected Outcome** |
| a | Enter valid student ID |  |
| b | Invalid data entries (e.g. wrong student ID) |  |
| d |  |  |