Simon Fraser University
School of Computing Science

Cmpt 275 Project

Date: 2015-07-21

Team name: Eleven ®

Project Deliverable #: 7

Project Deliverable Name: Test Cases Document

Phase(s) covered by this deliverable: White Box Testing, Black Box Testing

Phase leader(s): Janice Sargent

Team members and Student #:

Yinglun Qiao	301191776
Janice Sargent	301160485
Zhi Cheng	301184486
Susan Hamilton	301209641
Seong Jun Kim	301154246
Nari Shin	301178863
Fan Liu	301168674
Te Lun Chen	301200832
Roy Chan	301202770

Grade:

Revision History

Revision	Status	Publication/ Revision Date	By:
1.0	created test case for black box testing	*July 21, 2015*	Janice Sargent
1.1	added purpose, inputs, and subset to blackbox testing	*July 22, 2015*	Nari Shin
1.2	formatted and edited black box testing formatted white box testing table of contents added	*July 23, 2015*	Susan Hamilton
1.3	added sections 1-4 to white box testing	*July 23, 2015*	Susan Hamilton Nari Shin
1.4	Added sections 5 and 6 to white box testing Format addition to black box testing - added 2	*July 23, 2015*	Susan Hamilton
1.5	Added equivalent classes for black box testing	*July 23, 2015*	Janice Sargent
1.6	Added demonstrate test case	*July 24, 2015*	Janice Sargent
1.7	Edited table of contents, entire document for clarification	*July 25, 2015*	Nari Shin
2.0	Revision	*Aug 2 ,2015*	Fan Liu

2.1	Fixed decision paths in the list of test cases as suggested by Nethangi Added purpose of each test case in the list of test cases as suggested by Nethangi	*Aug 2, 2015*	Janice Sargent
2.2	Added Getting Started Section - software requirements	*Aug 3, 2015*	Janice Sargent
2.3	Added Sections to Table of Contents. Worked on the Post Mortem. Added a bug case.	*Aug 4, 2015*	Susan Hamilton
2.4	Added hardware requirements	*Aug 5, 2015*	Janice Sargent

Table of Contents

White Box Testing

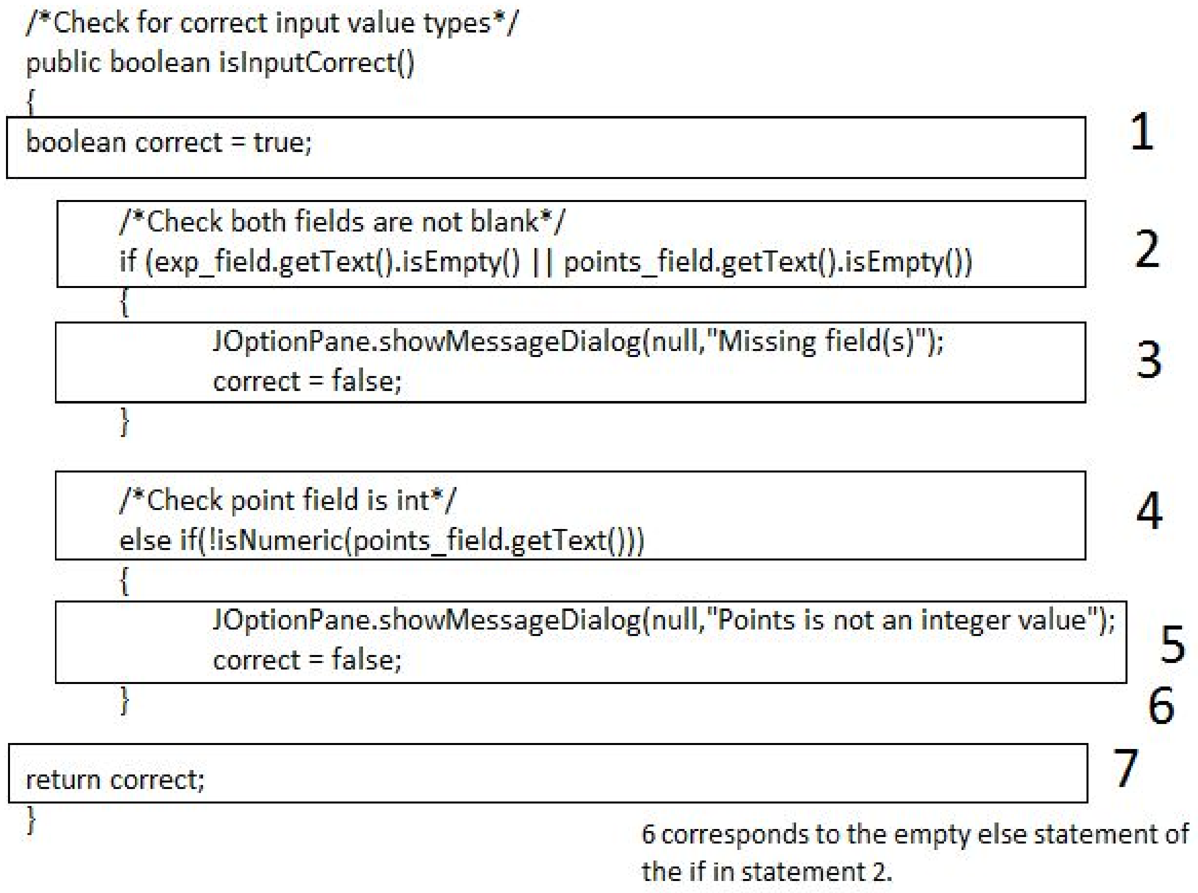
1 – Purpose

Describe the purpose of the method

The point of this method is to ensure that when creating a rubric that both expectation and points fields are not left blank and that the points field is an integer.

2 – Numbered Blocks

Analyze the code for the method and break it into appropriate numbered blocks

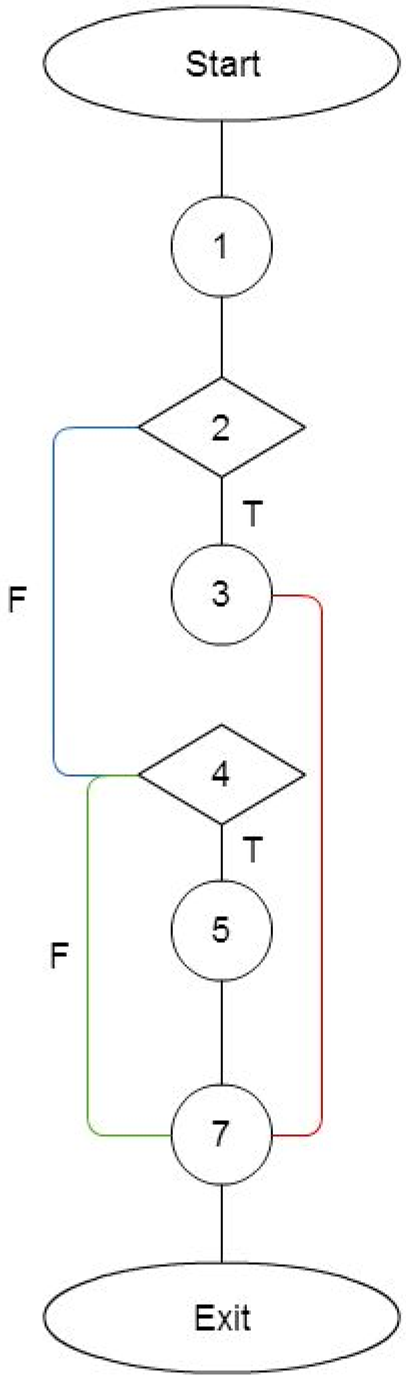


```
/*Check for correct input value types*/
public boolean isInputCorrect()
{
boolean correct = true;

        /*Check both fields are not blank*/
        if (exp_field.getText().isEmpty() || points_field.getText().isEmpty())
        {
                JOptionPane.showMessageDialog(null,"Missing field(s)");
                correct = false;
        }

        /*Check point field is int*/
        else if(!isNumeric(points_field.getText()))
        {
                JOptionPane.showMessageDialog(null,"Points is not an integer value");
                correct = false;
        }

return correct;
}
```

6 corresponds to the empty else statement of the if in statement 2.

3 – Flow Chart

Develop a flow chart for your method based on the blocks of code the method was broken into in the previous step



4 – Types of Coverage

Comment on the types of coverage needed to test the method

Since our methods contains selection statements and no loops, we will use branch coverage to test all the branches of our method to test each possible outcome.

Branch coverage: since the isInputCorrect() method contains selection statements, we will select input values that will cause each branch in the isInputCorrect() method to be executed once.
The method has two selection statements: one to check if the fields are blank, and the other to check if the points field is an integer.

In order to execute the branches of the missing fields statement, input should be blank in both or either fields then both filled. In order to execute the branches of the “not an integer value” statement, input should be a string or a character (eg. 12abc or abcde) and then an integer value (eg. 12345). This will achieve 100% branch coverage.

5 – List of Test Cases
Develop a list of test cases that will adequately test the method. Do not fully develop each test case. ONLY list the coverage of each test and explain why the test is needed/useful

Test case id – 1B coverage: Start-a-b-c-d-g-Exit
Purpose: to test if expectation field is empty
Inputs: Expectation field is empty

Test case id – 2B coverage: Start-a-b-c-d-g-Exit
Purpose: to test if points field is empty
Inputs: Points field is empty

Test case id – 3B coverage: Start-a-b-c-d-g-Exit
Purpose: to test if both expectation and points fields are empty
Inputs: Both fields are empty

Test case id – 4B coverage: Start-a-b-c-f-g-Exit
Purpose: to test if the input of points field is an integer
Inputs: Both fields are completed, points field is a string (contains characters eg. abcde)

Test case id – 5B coverage: Start-a-b-c-d-g-Exit
Purpose: to test if both expectation and points fields are not empty and the input of points field is an integer, the method returns true correctly
Inputs: Both fields are completed, points field is all integers. (eg. 10)

Test cases 1B,2B, and 3B ensure that no combination of fields are empty.
Test case 4B ensures that after both fields are complete, the Points field is not entered with a string.
Test case 5B ensures that after both fields are complete, the Points field contains only integers and the method is successful.

6 – One Test Case
Choose one of the test cases and describe it in detail in terms of the test case format used in class

Test id - 3B
Test Purpose: Unit test isInputCorrect() using white box testing strategy (branch coverage: Start - a -b - c - d- g-Exit)
Requirement: #1.4.2
Inputs: both exp_field and points_field are empty
Testing Procedures:

1) create two JTextField object - exp_field and points_field in the test driver
2) set the both text fields to empty by invoking exp_field.setText(“”) and points_field.setText(“”)
3) call isInputCorrect()

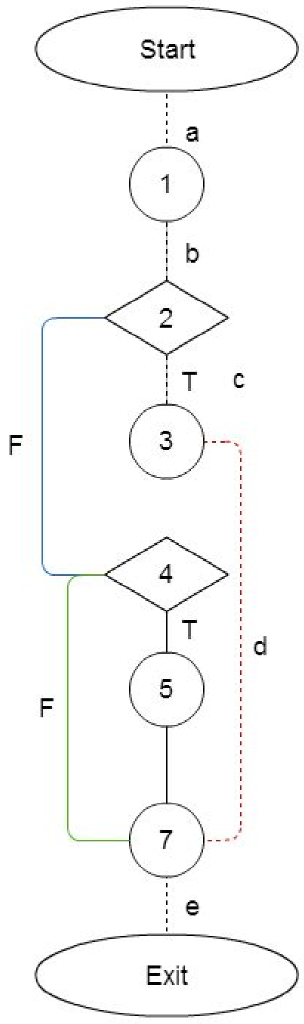
Evaluation: no step required
Expected behaviours and results: since both exp_field and points_field were set to empty, the method returned false

Actual behaviours and results: exp_field and point_field were blank, the method returned false as a result.

7 – Demonstrate Test Case

Demonstrate how your selected test case (input values) will help test the intended coverage.

The selected input values, exp_field = “” and points_field = “” are used to test the branch coverage. These inputs will help us test the intended coverage by testing our first if statement in block 2. Since both fields are blank the if condition should be true therefore testing one branch of our method.
The method isInputCorrect() is invoked, return value of false is expected.
Steps are as shown in the control flow chart below:



a) go to 1 and initialize local variables

b) go to 2 and execute 2 to check if any field is empty

c) since our selected input values for exp_field and points_field are set to empty, the if-statement returns true. The control flow then goes to 3 and executes the statements in 3 (the variable ‘correct’ is set to false)

d) since the if-statement in 2 is true, block 4, 5, and 6 are skipped, the flow then goes to 7 and executes it

e) the if-statement in 2 is false, the control flow goes to 4 and executes it

f) the if-statement in 4 is false, the control flow goes to 7 and executes it

g) the method returns false/true according to the result from d/f and exits

In this test case, the actual return value matches our expected return value.

Black Box Testing

1 – Purpose, Parameters, and Preconditions

Describe the purpose of the method, its parameters (arguments), and any preconditions necessary for successful execution of the method

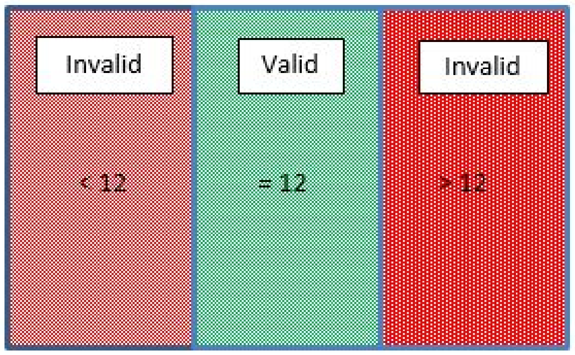
Method: testID(int id) - the method verifying if employee ID is a 12 digit number. It takes one parameter, an id, specified as integer.

Purpose: To determine whether the inserted employee ID is a valid 12 digit number. The method will use several representative input values to ensure that the employee ID fits all the requirements of containing only integers and having exactly a length of 12 digits, no more or less. If the employee ID fits the requirement of being a 12-digit number, the test case passes.
Parameters: “int id” will contain a 12-digit number.
Preconditions: The input of the parameter must be an integer in order to execute the method successfully.

2 – Equivalence Classes

Describe the equivalence classes for each of the inputs and preconditions of this method

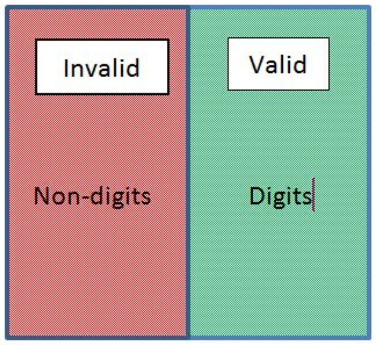
<u>Equivalence classes for employee ID</u>
Preconditions: input is an integer
There are 3 equivalence classes for range of values n:
- one valid equivalence class (program accepts n = 12)
- two invalid equivalence classes (n > 12 or n < 12)



Precondition: Input is an integer

There are two equivalence classes for the value of n:

- one valid equivalence class (program accepts integers - all digits)
- one invalid equivalence class (n is a string - non digits)



3 – Representative Input Values

Describe how you would choose representative input values to test this method

In order to test this method we would select input values that test for every possible case that a user may select.
We can have an n-digit number where n can be any length.
We know that in order to have a successful execution we need to enter a 12 digit number, so n=12 for our only valid input.
Other possible inputs would be a n-digit number where n<12 or n>12 but n != 12.
Therefore choose one value from each equivalence class for n.
Also choose the boundary values for each equivalence class.

Invalid	**Valid**	**Invalid**
<12	12	>12

e.g. choosing representative values n = 6, 12, 14
choosing boundary values n = 11, 12, 13
test values n = 6,11,12,13,14

We may also have an input that does not contain only digits. We may have an input that consists of characters (abcd), or a combination of digits and characters (a2b3c4).

Invalid	**Valid**
Non-digits	Digits

e.g. choosing representative values n = abcde, 12
test values n = abc123, abcd, 12

All Possible Inputs:
<12 digit number
12 digit number
>12 digit number
a string of characters of length n
a string of characters and numbers of length n

4 - Test Cases
Briefly describe how you would use the representative values to construct a set of test cases to test the method. DO NOT develop these test cases beyond describing them in terms of the values of the parameters and the preconditions.

First, test with all valid test cases and preconditions. After, test varying invalid test cases and invalid preconditions.
Precondition = Input is an integer.
N represents length of input.

Proposed Subset:
1. Variable within valid range, precondition true
(n = 12, all integers)
id = 123456789012, (int)id = true

2. Variable within valid range, precondition false
(n = 12, all characters)
id = abcdefghijkl, (int)id = false

3. Variable within valid range, precondition false
(n = 12, integers+characters)
id = 12345678901a, (int)id = false

4. Variable invalid range, precondition true
(n = 6, all integers)

id = 123456, (int)id = true

5. Variable invalid range, precondition false
(n = 6, all characters)
id = abcdef, (int)id = false

6. Variable invalid range, precondition false
(n = 6, integers +characters)
id = abc123, (int)id = false

7. Variable invalid range, precondition true
(n = 11, all integers)
id = 12345678901, (int)id = true

8. Variable invalid range, precondition false
(n = 11, all characters)
id = abcdefghijk, (int)id = false

9. Variable invalid range, precondition false
(n = 11, integers +characters)
id = abcdef12345, (int)id = false

10. Variable invalid range, precondition true
(n = 13, all integers)
id = 1234567890123, (int)id = true

11. Variable invalid range, precondition false
(n = 13, all characters)
id = abcdefghijklm, (int)id = false

12. Variable invalid range, precondition false
(n = 13, integers +characters)
id = abcdef1234567, (int)id = false

13. Variable invalid range, precondition true
(n = 14, all integers)
id = 12345678901234 , (int)id = true

14. Variable invalid range, precondition false

(n= 14, all characters)
id = abcdefabcefab, (int)id = false

15. Variable invalid range, precondition false
(n = 14, integers+characters)
id = abcdef12345678, (int)id = false

5 – Single Test Case
Select a single test case from those discussed in the previous item.
Test case chosen from above is subset #1 where i = 123456789012.

6 – Test Case Format
Describe your test case using the test case format seen in class.

Test case id – 1

Test purpose: Unit test UserAccount class method testID(int id) using black box testing strategy.

Requirement: #2.2.1.3

Inputs: i = 123456789012

Testing procedure:
- Create an object of UserAccount class type
- Prompt user to enter employee id
- setEmployeedId(id) invokes testId(i = 123456789012)

Evaluation: Show database

Expected behaviours and results: Id is a 12-digit number which fulfills both requirements of being the correct length and consisting of only integers. Method should validate the employeeID.

Actual behaviors and results: Id passes both test cases employeeId is valid and therefore validated for setEmployeedId(id) method when it invokes testid(id)

Post Mortem

1 – What We Learned: Project Management

Discuss what you have learned about project management and time estimation. As a part of your discussion, compare your original estimate of the time needed with the actual time needed and draw some conclusions. For example, which phase was the longest? Shortest? Do these observations surprise you?

Project Management is made up many parts that need to function well together for the success of the project. There are management skills; communication skills; listening skills; problem solving between members of the group, between the client and the group, and within the code; decision making conflict resolution skills, giving and receiving feedback, and most importantly leadership skills to ensure that all other skills are functional. We were surprised by how much lead up there was to the actual programming. Right at the beginning our group was a bit thrown off by how long it took to go through the rough list of requirements and form them into a formal list of requirements to ensure that there was no holes in understanding between the client and the software creators. It also meant that we spent a lot of time rewriting and suggesting scenarios to encompass the requirements and to find where there were unclarified sections of the requirements are. After this we still had to make class diagrams and come up with testing methods. Finally after this long leed up we spent the last 3 or 4 weeks, out of 13, on the programming. So in summary, it was shocking how much time of project management was spent on ensuring the requirements were clear and then planning out the code, but because of how huge our group is and because of the the complexity of the software we were producing. Also, it was interesting having the dynamic of each member of the group having a different management role: communication manager, configuration manager, meeting manager, and minutes manager along with each person taking a phase role. These roles allowed for each one of us to take charge in different aspect of the project whether it was communication, planning, or creating uniform documentation.

Time estimation can be a very tricky thing to establish for anyone and, for many students going into CMPT 275, we are aware of how this class takes a lot of time. Upon initially beginning this course I knew that the most time consuming section would be the actual programming and the documents would take a little less time, but the documents actually took more time, but there was an interesting dynamic that would throw this truth out. The documents would take a lot of time to understand and to research, with lots of time spent on group discussion and upon waiting for different members to contribute to the document with lots of rereading and editing with more communication and debate. This took a long time, but it was something that you could work for a bit on and then continue with other things. Programming, though it took less time, it required more diligence and coordination with meeting. It meant that you could not just work for a 1 hour block on the code, but instead more like a 4 hour block. As it also took place at the end of the term it ended up clashing the most with other courses. So even though this section took the least amount of time it ended up being the part where we fell the most behind. It was shocking because I felt that the coding would be the most independent where you could work in your own time on finishing you section by the deadline. Also, as a last note on time estimation we found that it was best to assume everything would go wrong and take forever, forcing you to start earlier rather than to wait to the last minute, because you never know if a bug will take a quick fix or a whole rewrite of the method.

2 – What We Learned: Testing

Discuss what you have learned about testing.

Testing, as we first learn it, was using a bunch of pint statements, using breakpoints, and debugging. However in this course we learned about the approaches of white box and black box testing. Both test methods use the technique of trying to cause failures in the software by checking the boundary cases and random use cases of the variable are covered. Specifically, black box testing applies test cases by knowing what the expected results or behaviour of the system are. So this would be looking for test values which are boundary values; ie if your input value is required to be a 12 digit number then boundary cases would be 11,12, and 13; and representative values would be a value less than 12 such as 6, a value greater than 12 such as 14, and a valid number such as 12 (the only one in this example). We would have test cases each of this length with also all integer values, all characters, and a mix of the two, two test the input and the respective output completely and thoroughly. White box testing, on the other hand uses another approach which is by using knowledge of the internal structure of the code to ensure that in certain test cases each part of the code flows correctly.

This is done by using breaking up the method into blocks which are numbered and then creating a flowchart showing how each part of the code triggers and in what order, ie. which blocks call other blocks, and showing this flow in a list of test cases to ensure that the code all runs correctly in a given situation. In a sense white box testing is how you would test your own code after writing it, while black box testing is how you or another member of your group would test the functionality success of the code. Both are integral to the success of creating more successful software.

3 – What We Learned: Team Setting

Discuss what you have learned about working in a team setting.

Working in a group of 9 was a daunting situation, but by taking several measures, as suggested in class, we were able to work the numbers in our favour so that we could obtain a more cohesive whole. As was suggested, we made use of, upon first forming our group, assigning management roles. These roles were communication manager, configuration manager, meeting manager, and minutes manager along with each person taking a phase role. This allowed for us to address our main concerns to the respective management to ensure that they were taken care of; such concerns being any confusion over the task at hand or with what was expected of them. By assigning roles it also ensured that no one member was overburdened with work throughout the duration of the project.

Though all this did help to create a cohesive group we still had some problems. The problem which was the most cumbersome to our groups success was communication. We did assign parts to members of the group, often ensuring members were grouped up so that should a member get stuck they could still turn to someone else familiar with the code to ask for help. With this being said, however, we ended up having troubles having these groups meeting up to ensure that the work was done, where in one instance a member misunderstood another and thought that their section had been completed when in fact it was only a small section of it which had been completed. Another situation had a group member accidently complete another members part and then leaving the other member with nothing to commit.

From this our group has learned that the most important thing to establish right from the get go is communication. This may include ensuring that members check their emails, skype, or other social media to ensure they have seen any updates or questions being made by other members, but it also means that keeping minutes or notes on what was decided are formally made to ensure that every member has a clear understanding of what is expected of them. One tool that our group utilized in this sense was google docs where we would post the requirements of the upcoming iteration due and would show who was assigned to each part and using a colour system to show which parts were complete, unfinished, bugged, and needed further clarification. So in summary the most vital lesson our group established is the importance of communication.

Getting Started

1 – Software Requirements

The streamlined grading system is limited to Windows operating system (Windows 7 and above). . The system, written in java, will be interfaced using java swing components and Microsoft sql server 2012 is used to store the database on the server. In addition, the following softwares are used for the system:

- The rs2xml.jar is used to help to display content in sql database table.
- The sqljdbc4.jar is used to connect to sql database.
- NetBeansIDE 8.0.2 is used to install and run this grading system.
- Java 1.8.0_40 is required to install and run this grading system.
- JDBC driver that provides database connectivity through the standard JDBC application program interfaces (APIs) available in Java Platform.

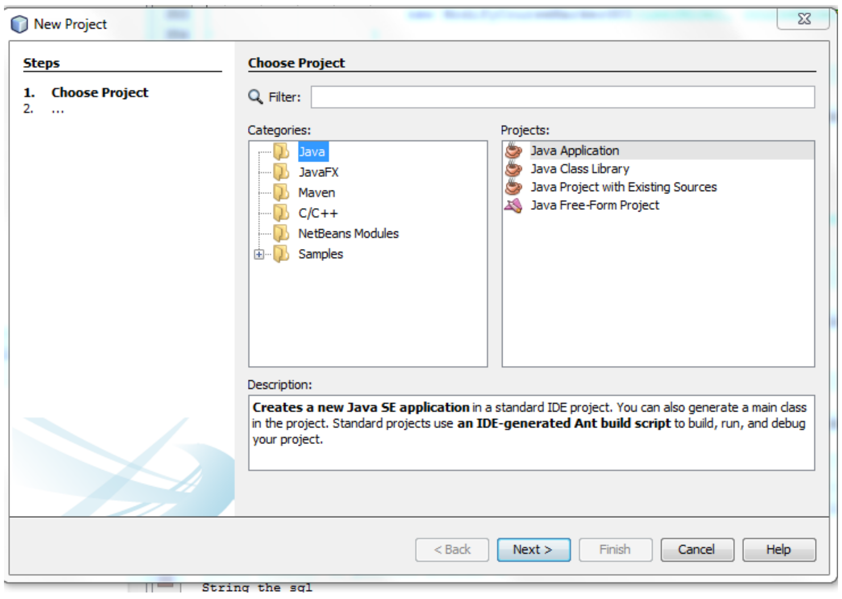
2 – Hardware Requirements

Operating Platform (Windows)
Recommended: 1.4GHz processor, 2GB RAM, 20GB disk

Database server
Recommended: 3 GHz processor, 4GB RAM, 100 GB RAID4 + disk

3 – Technical Installation and Compilation Steps

1. Open NetbeansIDE
2. Click “File” Menu
3. Choose “New Project... ”
4. Then this screen (Image A) will pop up



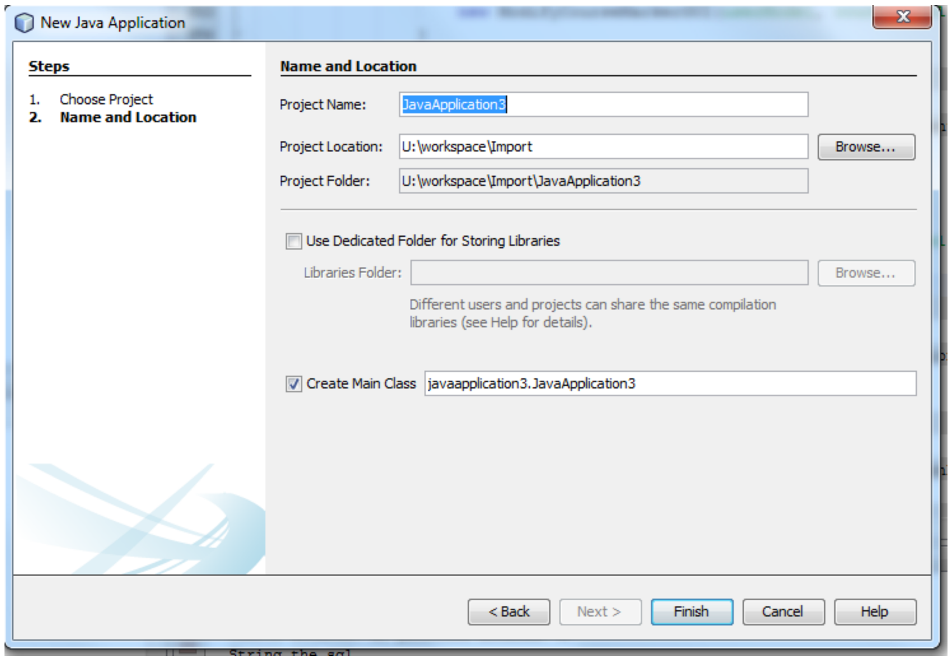
Description:

Creates a new Java SE application in a standard IDE project. You can also generate a main class in the project. Standard projects use **an IDE-generated Ant build script** to build, run, and debug your project.

Image A

5.Choose “Java” on the left and Choose “Java Application” on the right

6.Click “Next>” button


New Java Application

Steps
1. Choose Project
2. **Name and Location**

Name and Location

Project Name: JavaApplication3
Project Location: U:\workspace\Import
Project Folder: U:\workspace\Import\JavaApplication3

[ ] Use Dedicated Folder for Storing Libraries
Libraries Folder:
Different users and projects can share the same compilation libraries (see Help for details).

[✓] Create Main Class javaapplication3.JavaApplication3

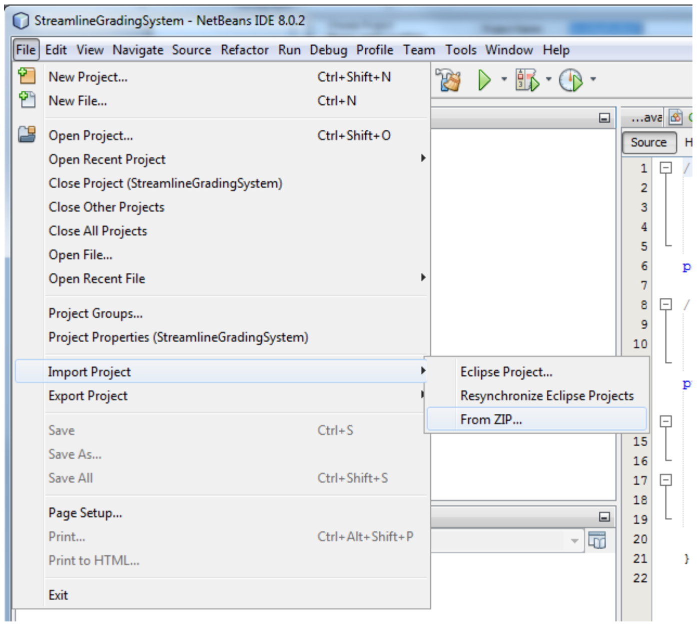
Image B

7. The "New Java Application" screen (Image B)will pop up. User type in the name "StreamlineGradingSystem" of the project in the Project Name field.

8. User need choose a local disk folder to store the grading application.

9. Click "Finish" button.

10.Click "File" Menu and then choose "Import project" and then choose "From ZIP…"(Image C)

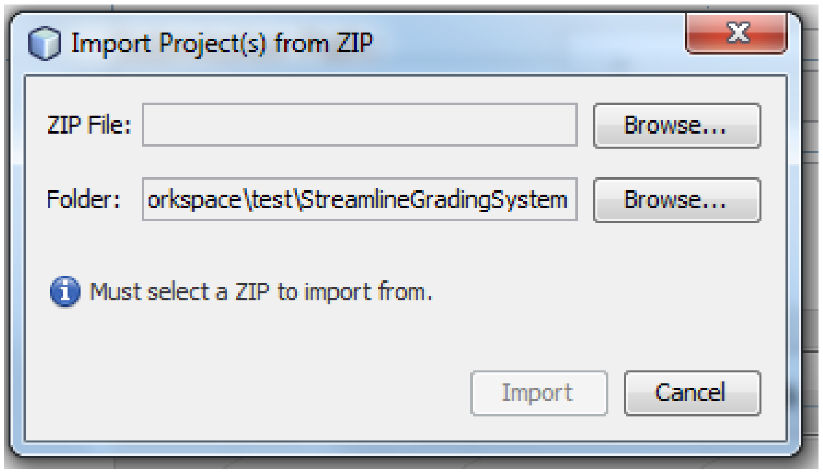

StreamlineGradingSystem - NetBeans IDE 8.0.2

File Edit View Navigate Source Refactor Run Debug Profile Team Tools Window Help
New Project... Ctrl+Shift+N
New File... Ctrl+N
Open Project... Ctrl+Shift+O
Open Recent Project
Close Project (StreamlineGradingSystem)
Close Other Projects
Close All Projects
Open File...
Open Recent File
Project Groups...
Project Properties (StreamlineGradingSystem)
Import Project
Export Project
Save Ctrl+S
Save As...
Save All Ctrl+Shift+S
Page Setup...
Print... Ctrl+Alt+Shift+P
Print to HTML...
Exit
Eclipse Project...
Resynchronize Eclipse Projects
From ZIP...
Source

Image C

11. The “Import Project(s) from ZIP” in the zip file field choose “Browse…” button and then select the path to the source code and click to the zip type file. Then click “Import” button and wait (Image D)

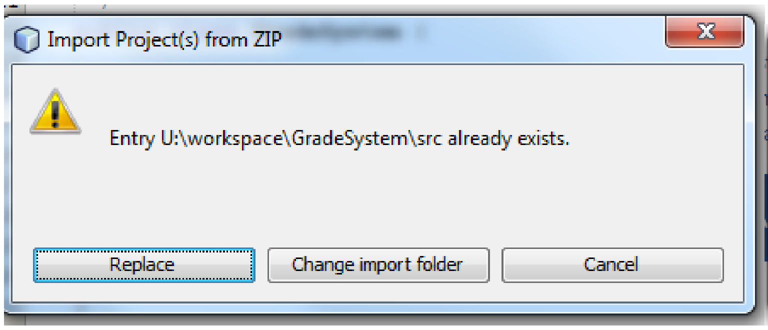
12.The Message box “No NetBeans projects add” may pop up. Just click “ok” button.



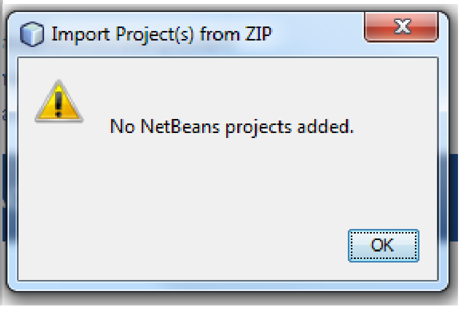
Folder: orkspace\test\StreamlineGradingSystem

Must select a ZIP to import from.

Image D



Entry U:\workspace\GradeSystem\src already exists.



No NetBeans projects added.

Image H

Image I

13.During importing process the error message “Empty ... already exists”(Image H) may pop up just click “Replace” button

14. During importing process the error message “No Netbeans projects added” (Image I)may pop up just click “ok” button

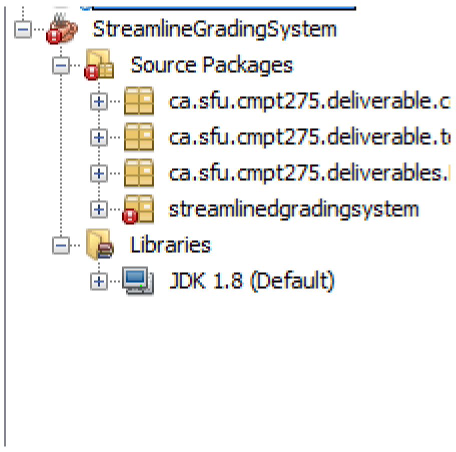


Image E

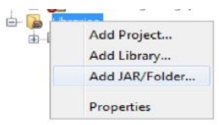


Image F

15. In the Libraries part(Image E), right click “Libraries” choose “Add JAR/Folder”(Image F).

16.Add “sqljdbc4.jar” and “rs2xml.jar” to the project

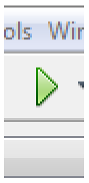
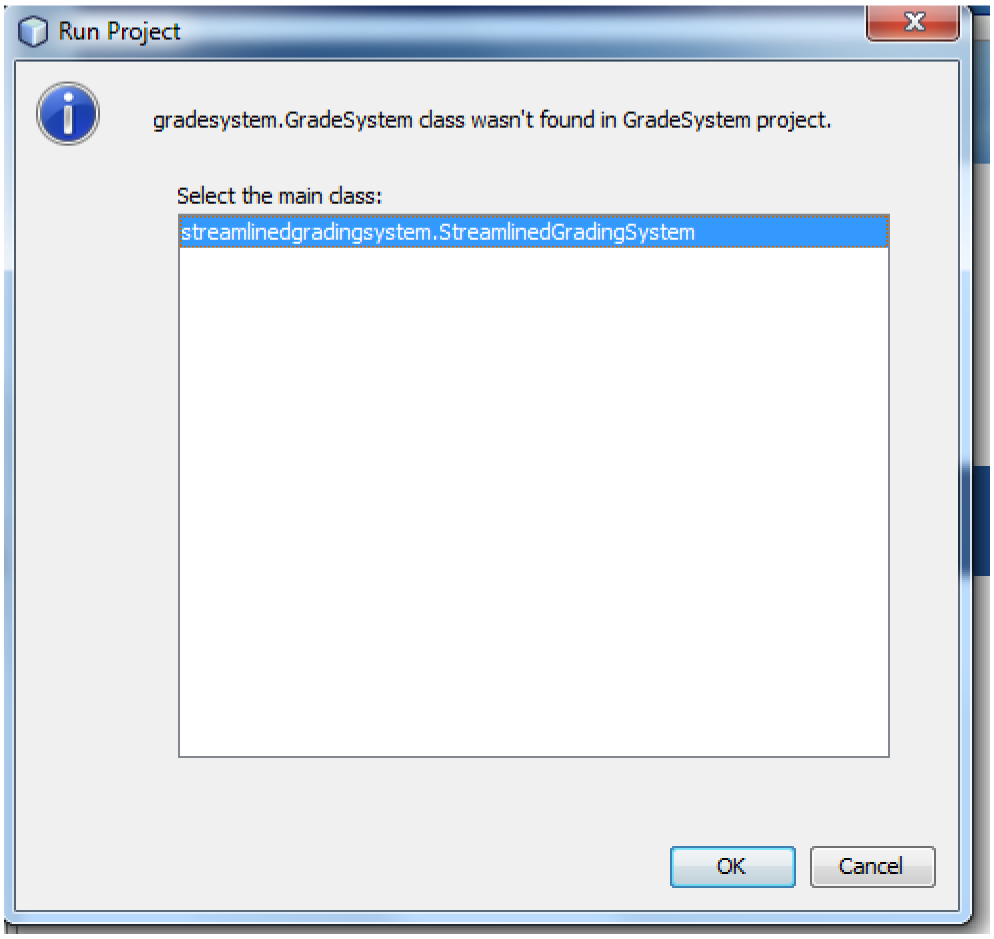


Image G

17.Click “run”(Image G) button to NetBean build and run the project.



gradesystem.GradeSystem class wasn't found in GradeSystem project.

Select the main class:

streamlinedgradingsystem.StreamlinedGradingSystem

Image J

18. When user click “run” button, this message(Image J) box will pop up,

Just press “ok” button

19. In the local disk, user need create folder named “CMPT 275”in U(SFU home network drive) directory ,under that folder a sub-folder called ”files” and under “files” folder contain all students’ submission and instructor solution file.

Known Bugs

- Because of how the database was made (can be changed) a student number for this version of the software must be less than 2,147,483,648 since it's a signed integer value that is only 32 bit since.