ICT167 Principle of Computer Science

**Table of Contents**

[**1.** **Title** 4](#_Toc159980461)

[**2. Requirements/Specification** 5](#_Toc159980462)

[**3. User Guide** 6](#_Toc159980463)

[**4.** **Structure/Design/Algorithm:** 12](#_Toc159980464)

[**5.** **O-O design concepts** 18](#_Toc159980465)

[**6. Limitations** 21](#_Toc159980466)

[**7. Testing** 22](#_Toc159980467)

# **1.** **Title**

**Change Management System Documentation:** This document, authored by Keith Neo Kai Si on February 26, 2024, outlines the functionalities of the Change Management System. It encompasses five Java files: CoinDenominationCountry.java, CoinCountNode.java, CoinNodeList.java, Change.java, and Client.java. The system aims to manage coin count data and calculate change efficiently. CoinDenominationCountry.java defines coin denominations for specific countries. CoinCountNode.java represents a node in a linked list storing coin counts, while CoinNodeList.java manages linked lists of coin count nodes. Change.java handles change calculation for individuals, allowing manual input or use of hardcoded test data. Client.java serves as the main program, enabling user interactions and displaying results. These classes form a comprehensive system for managing and processing change-related data.

# **2. Requirements/Specification**

The program is designed as a change management system that allows users to enter data about their data and their coin holdings and perform various operations on this data. Users can enter data manually by specifying a person’s name and corresponding coin balance or use hard-coded test data for convenience. The program calculates the optimal combination of coins needed to form each person’s exchange amount, considering the available coin values set per country. Next, a breakdown of each denomination’s change and the total number and amount will be displayed. The valid coin amount is assumed to be a multiple of 5, and the name consists of one word. The expected output includes detailed information about each person’s change, including their name, total change amount, and change breakdown by denomination. In addition, the program could find the person with the largest and smallest amount of coins, calculate the total number of coins of each denomination for all people, and find the total amount, which is the sum of all coins.

# **3. User Guide**

**Complication and Execution:**

1. **IDE:**  [Eclipse IDE 2024-03](https://www.eclipse.org/downloads/packages/installer) is used for this program.

2. **Compilation:** To begin, launch Eclipse and proceed to import the Java project. Navigate to the “File” tab, then select “New” followed by “Java Project.” There will be a pop-up menu that will require a name for the Java project. Be sure to deselect the default location option and specify the source folder's location (containing the Java files). Additionally, ensure that the “Create module-info.java info” option is unchecked. Finally, click on “Finish” to complete the process.

A screenshot of a computer

Description automatically generated

3. **Execution:** First, right-click on the newly created Java project to execute the Java Project. Then, click the “Run as” dropdown menu and select “Java Application”. This action will execute the Java application.

A screenshot of a computer

Description automatically generated

**Using the Program:**

1. **Data Input Choice:**

* Upon execution of the program, the screen will display student information along with two options: users can either input their data or use pre-defined test data.

**A screen shot of a computer

Description automatically generated**

* If the user chooses option 1 (User Define Data), they will initially be prompted to input the person’s name. If the name contains whitespace or is left blank, the system will prompt “Invalid name. Try again.” Following this, the user will be asked to enter the coin value for that person. If the input is empty or negative, or it’s not a multiple of 5, the system will prompt “Please provide a valid input” or “Incorrect coin value. Must be a multiple of 5 and cannot be less than 0” accordingly. After valid inputs are provided, the system will inquire if there are additional people to enter.

**A screenshot of a computer program

Description automatically generated**

* If the user selects option 2, the hard-coded test data will be directly loaded into the ChangeArray.

**A screen shot of a computer

Description automatically generated**

1. **Main Menu:**

* Following the data input section, the main menu will be presented, offering 6 options for the user, each corresponding to various functionalities as shown below.

**A screen shot of a computer

Description automatically generated**

1. **Selecting Options:**

* IF the user inputs nothing, letters, or unknown options, the system will prompt “Invalid input. Try again.” Or “Unknown choice. Try again.” Respectively, prompting the user to retry.

A screenshot of a computer

Description automatically generated

* If option 1 is selected, the program will prompt the user to input a person’s name. Subsequently, it will display the person’s name, their total change amount, and the breakdown of their change for each denomination. Additionally, if the user enters a name not found in the Change array, the system will display “Not Found” instead of user details.

**A screenshot of a computer program

Description automatically generated**

**A screen shot of a computer

Description automatically generated**

* If option 2 is selected, the system will search for and present the person’s name, their coin change amount, and each denomination with the highest amount of coin change in the change array.

**A screen shot of a computer

Description automatically generated**

* If option 3 is selected, the system will locate and display the person’s name, their coin change amount, and each denomination with the smallest amount of coin change in the change array

**A screen shot of a computer

Description automatically generated**

* If option 4 is chosen, the system will compute the total amount of each denomination for every person in the change array and present the results. It’s important to note that the calculation of each denomination does not involve summing up all coin change amounts and then calculating; instead, it calculates each denomination for each person separately.

**A screen shot of a computer

Description automatically generated**

* If option 5 is chosen, the system will compute each person's total amount of coin change and display the results.

**A screen shot of a computer screen

Description automatically generated**

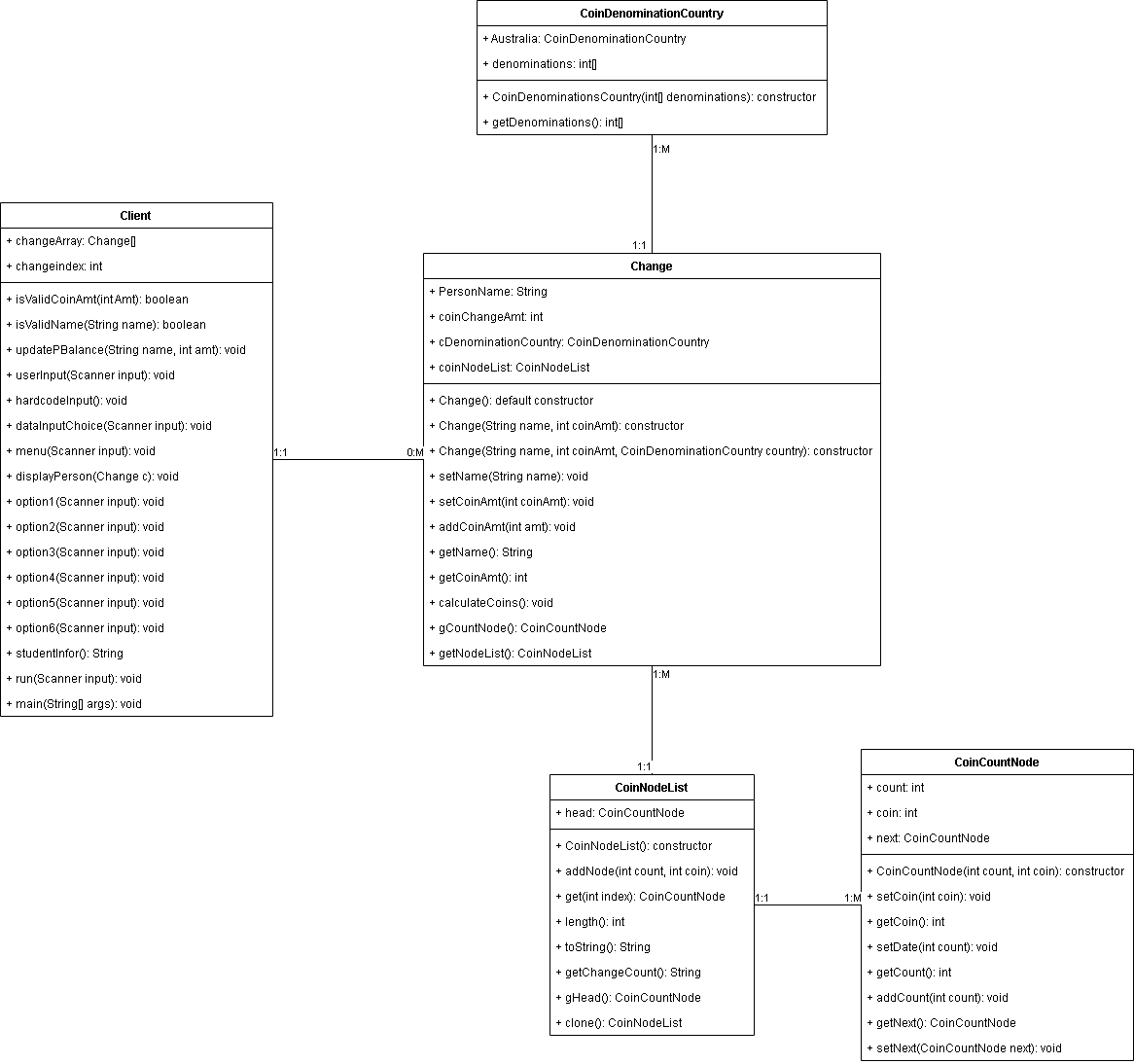
* If option 6 is selected, the program will send a farewell message and terminate.

**A screen shot of a computer

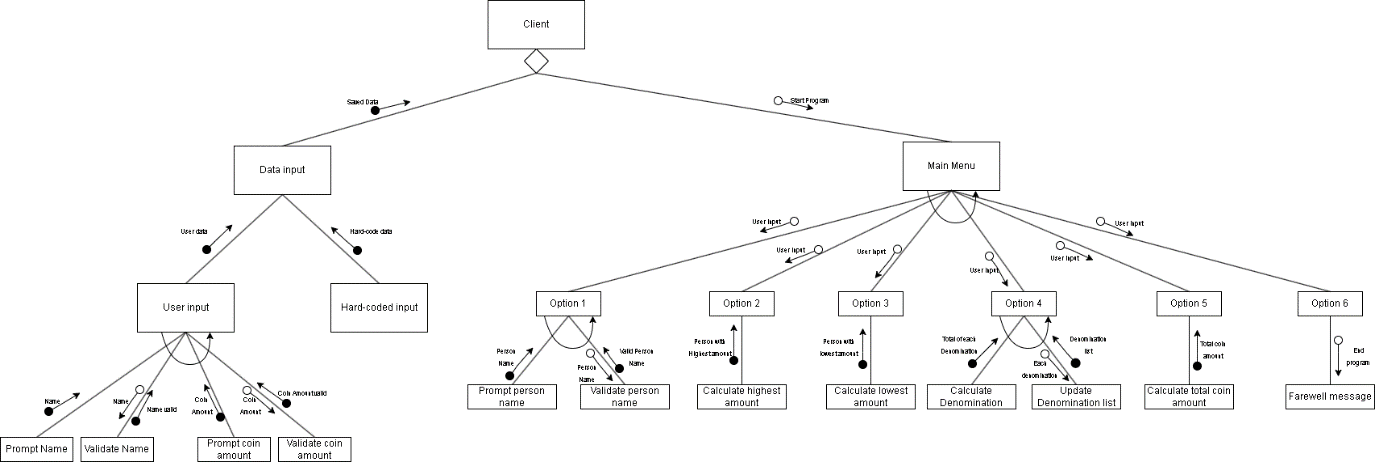
Description automatically generated**

# **4.** **Structure/Design/Algorithm:**

**UML for the Client program**

****

**Structure chart for the Client program**



**CoinCountNode class**

The CoinCountNode class is an essential element in a linked list structure tailored to manage coin count data efficiently. Once instantiated, each node encapsulates necessary details, including the number of coins of a particular denomination and the denomination itself. Using methods such as setCoin() and setData() can allow adjustment of the denomination or increase the desired quantity.

Moreover, the class provides additional functions, such as addCount(), that can dynamically adjust the number of coins in a node, facilitating scenarios where the amount of coins fluctuates. Notably, nodes are connected by references, facilitated by methods like setNext(), allowing for constructing a linked list. Methods like getNext() will enable the system to navigate these connections, making it easy to traverse the list, traverse all nodes, or access a specific node based on an index.

The CoinCountNode class, therefore, serves as the basis for organizing and managing coin information within a linked list structure, providing a robust framework for a variety of information that requires a structured and efficient processing of coin data.

**CoinNodeList class**

The CoinNodeList class is a versatile tool for efficiently managing coin data within a linked list structure. Upon instantiation, the class initializes an empty list that forms the basis for storing coin count information. The addNode() function makes adding nodes to the list manageable. This function adds a new coin node with the specified number and denomination.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | Pseudocode explanation: addNode() | | | | | | | |
| **1.** | **FUNCTION** addNotes(count, coin) | | | | | | | |
| **2.** |  | **IF** head is **NULL** | | | | | | |
| **3.** |  |  | **SET** head to **newCount** | | | | | |
| **4.** |  | **Else** | | | | |
| **5.** |  |  | **INITIALISE** current to head of the list | | | |
| **6.** |  |  | **WHILE** current.getNext() is not **NULL** | | | |
| **7.** |  |  |  | **SET** current to next node | | |
| **8.** |  |  | **END WHILE** | | |
| **9.** |  |  | **SET** next node to current | | |
| **10.** |  | **END IF** | | |
| **11.** | **END FUNCTION** | | | | | | |

The get() function allows the system to retrieve nodes by index, making accessing specific elements in a list easy.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Pseudocode explanation: get() | | | |
| 1. | **Function** get(index) | | | |
| 2. |  | **If** index is less than 0 | | |
| 3. |  |  | **Return** null | |
| 4. |  | **End If** | | |
| 5. |  | **Initialise** coinNodeIndex to 0 | | |
| 6. |  | **Initialize** current to head | | |
| 7. |  | **While** current is not **Null** | | |
| 8. |  |  | **If** coinNodeIndex equals index | |
| 9. |  |  |  | **Return** current |
| 10. |  |  | **End If** | |
| 11. |  |  | **Increment** coinNodeIndex by 1 | |
| 12. |  |  | **Move** current to next node | |
| 13. |  | **End While** | | |
| 14. |  | **Return** null | | |
| 15. | **End Function** | | | |

Additionally, this class provides a function to calculate the length of a list via the length() function, allowing knowledge of the size of the data structure. The toString() function is designed to iterate through each node in a linked list, starting from the head node. Create a string representation for each node, considering the coin value and the corresponding count. The denomination of a coin is determined based on whether the value stored in the node’s coin attribute is divisible by 100, indicating whether it represents a denomination in dollars or cents. These denomination and count pairs are formatted and added to the nodeListString variable. On the other hand, the getChangeCount() function works similarly to toString() by iterating through each node in the linked list. However, the difference is that string representations are only created for non-zero coin counts. Instead of including all denominations and count pairs, only non-zero counts are selectively added to the coinChangeString variable.

|  |  |  |  |
| --- | --- | --- | --- |
|  | Pseudocode Explanation: toString() | | |
| **1.** | **Function** toString() | | |
| **2.** |  | **INITIALISE** current to head | |
| **3.** |  | **INITIALISE** an empty string nodeListString | |
| **4.** |  | **While** current is not **null** | |
| **5.** |  |  | **If** coin value is multiple of **100**, format as dollar |
| **6.** |  |  | **Otherwise**, format as cents |
| **7.** |  |  | **Append** coin representation and count to nodeListString |
| **8.** |  |  | **If** current has next node, append newline to nodeListString |
| **9.** |  |  | **Move** to next node |
| **10.** |  | **End While** | |
| **11.** |  | **Return** nodeListString | |
| **12** | **End Function** | | |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Pseudocode Explanation: getChangeCount() | | | |
| **1.** | **Function** getChangeCount() | | | |
| **2.** |  | **INITIALISE** current to head | | |
| **3.** |  | **INITIALISE** an empty string coinChangeString | | |
| **4.** |  | **While** current is not **null** | | |
| **5.** |  |  | **If** coin value is multiple of **100**, format as dollar | |
| **6.** |  |  | **Otherwise**, format as cents | |
| **7.** |  |  | **If** count of current node **Is Not** zero | |
| **8.** |  |  |  | **Append**  coin representation and count to coinChangString |
| **9.** |  |  | **If** current has next node, append newline to nodeListString | |
| **10.** |  |  | **Move** to next node | |
| **11.** |  | **End While** | | |
| **12.** |  | **Return** coinChangeString | | |
| **13.** | **End Function** | | | |

Additionally, the gHead() function allows the system to retrieve the head of the node of the list of nodes, and the clone() function is responsible for creating a deep copy of the current linked list. To accomplish this, this method initializes a **new instance** of CoinNodeList called clonedList to store a cloned version of the original list. Next, check that the original list is empty by checking that the **head** is not null. If the original list is not empty, the function loops through each node in the original list, creating a new CoinCountNode for each node and adding it to the clonedList using the addNode() function. Once all the nodes

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Pseudocode Explanation: clone() | | | |
| 1. | **Function** clone() | | | |
| 2. |  | **Create**  new CoinNodeList | | |
| 3. |  | **If**  head of current is not null | | |
| 4. |  |  | **Initialize** current to head | |
| 5. |  |  | **While**  current is not **null** | |
| 6. |  |  |  | **Add** new node to clonedList from current node |
| 7. |  |  |  | **Move** current to next node |
| 8. |  |  | **End While** | |
| 9. |  | **End If** | | |
| 10. |  | **Return** clonedList | | |
| 11. | **End Function** | | | |

**CoinDenominationCountry class**

This class is initialized with a predefined set of country-specific denominations, represented by a static instance of CoinDenominationCountry, like Australia, which lists the denomination in cents in descending order. The system can use the getDenomination() function to access these denominations, which returns an array of integers representing denominations in descending order. This encapsulated denomination set cannot be modified externally, ensuring data integrity and protecting the predefined denomination formed. This setup allows for easy access to coin denominations within the country and allows the simplistic addition of other country denominations by adding another country in the CoinDenominationCountry class.

**Change class**

The Change class facilitates the calculation and management of the coin denominations required to form a person’s change amount, considering the coin denomination of a particular country. It offers attributes such as “personName,” “coinChangeAmt,” “cDenominationCountry,” and “coinNodeList.” The constructor can be initialized with default values or the person’s name, coin amount, and country. Functions include setter and getter for attributes, addCoinAmt to update the change amount, and calculations to determine the best combination of coins for the change amount. It will return a detailed breakdown of the coin denominations by iterating over them and creating a coinCountNode object for each denomination. This process includes setting the change amount, calculating coins based on denomination, and getting the breakdown. This class provides flexibility and precision for specific country denominations and efficiently performs calculations of a coin denomination for a particular amount of coin.

The calculateCoin function is initialized by setting the remaining amount to the change amount and obtaining the coin denominations for the specified country. It then loops through each denomination to calculate the numbers needed to form the remaining amount. A CoinCountNode containing the estimated count is added to the list for each denomination. This process updates the remaining amount by subtracting the value of the coins until the balance is zero or all denominations are considered. The resulting “coinNodeList” provides a detailed breakdown of the optimal coin combinations needed to meet the change amount based on the country’s denomination.

|  |  |  |  |
| --- | --- | --- | --- |
|  | Pseudocode Explanation: calculateCoin() | | |
| 1. | **Function** calculateCoins() | | |
| 2. |  | **Initialize** remainingAmt to current change amount | |
| 3. |  | **Retrieve** array of coin denomination for the country | |
| 4. |  | **Reset** coinNodeList to new coinNodeList | |
| 5. |  | **For each** coin denomination in denomination array | |
| 6. |  |  | **Calculate** count for current denomination |
| 7. |  |  | **Add** new not to coinNodeList with calculated coin and denomination |
| 8. |  |  | **Update** remainingAmt by subtracting value from coins |
| 9. |  | **End For** | |
| 10. | **End Function** | | |
|  |  | | |

**Client class**

This class provides functionality to manage data input through user interaction or predefined test data. Users can interactively enter individual coin holdings via userInput or hardcoded test data for testing purposes. The menu system guides the user through various options, including displaying the denomination count for specific individuals, identifying those with the largest or smallest amounts, calculating total coins denomination coins, and determining the overall denomination total. Each menu option corresponds to a function that performs the selected action based on user input, ensuring a structured interaction flow. The validation methods validate the user input, ensuring that the name is valid, that that coin amount is a multiple of 5, and that the string can be converted to an integer. Error messages prompt users to correct invalid input to ensure smooth program interaction. The display method outputs information about the individual’s change amount and total coin count to the console in a formatted format. The program is started through a run function that handles data input selection and menu display, and the primary function initializes program execution by creating a scanner object for user input.

# **5.** **O-O design concepts**

**Open/Closed Principles:**

The Open/Closed Principle (OCP) is demonstrated in the solution by implementing the Change class. This class is designed to be open for extension but closed for modification. This is achieved by encapsulating the logic for calculating coin values into the ‘calculations ()’ function. This function can be extended to support additional coin value configurations without changing existing code. For example, a new country with a different coin denomination can be created and used with the existing Change class without changing the internal implementation. This adheres to the OCP by allowing the Change class to easily extend to support new functionality without changing existing code.

**Information Hiding:**

Information hiding is implemented through the solution to ensure that the internal details of each class are hidden from external users. For example, the CoinCountNode class encapsulates the details of each node in a linked list. It provides only the necessary methods, such as accessors and mutators, such as setCoin() and getCoin(), to interact with the node’s data. Similarly, the CoinNodeList class hides the complexity of managing a linked list of coin count nodes by providing high-level functions such as addNode() and get() for external use while keeping the internal implementation hidden. This promotes modularity and abstraction, allowing users of these classes to interact with them at a higher level without understanding their inner workings.

**Encapsulating**

Encapsulation is a crucial design principle demonstrated throughout the solution. Each class encapsulates its data and functions, providing only the interfaces needed for interaction while hiding internal implementation details. For example the Change class encapsulates details of an individual’s change amounts and corresponding coin values and provides methods such as calculateCoins() to interact with this data. Similarly, the CoinCountNode and CoinNodeList classes use a linked list structure to encapsulate the details of managing coin count data, providing only the functions needed to add, retrieve, and edit coin counts. This encapsulation ensures that each class maintains a clear and well-defined interface, promoting the codebase's modularity, reusability, and maintainability.

**Preconditions and Post-conditions:**

Preconditions and postconditions are enforced implicitly using addNode() and calculateCoins(), requiring specific conditions to be met before and after executing the function. For example, the addNode() method of the CoinNodeList class requires that the function expects valid parameters representing the number and denomination of coins to be added to the list. Similarly, the ‘calculateCoins()’ function in the Change class has a postcondition that guarantees the correct calculation of coin denominations based on the change amount. Defining and enforcing these preconditions and postconditions increases the robustness and reliability of the code, reduces the possibility of errors, and ensures that the system behaves predictably under various conditions.

**Private Variables**

Private variables are widely used in deployed solutions to limit direct access to internal data and ensure data integrity. For example, instance variables such as head in the CoinNodeList class and count, coin, and next in the CoinCountNode class are declared private to prevent direct modification by external classes. Instead, access to these variables is controlled by accessor and mutator functions such as get() and setNext(), which enforce encapsulation and information hiding. This prevents unauthorized changes to the internal state, reduces the risk of unintended side effects, and improves the reliability and maintainability of the entire codebase.

**Constructors**

Constructors are used in all classes to initialize the state of objects and ensure that they are properly initialized when an instance is created. For example, the CoinNodeList class provides a default constructor that initializes the head node to null, ensuring new class instances have an empty list. Similarly, the CoinCountNode class has constructors that accept initial values for the count and denomination of a coin node, making it easy to instantiate new nodes with the specified values. By providing a constructor, classes encapsulate initialization logic internally, prompting consistency and ease of use of the program that instantiates.

**Assessors and Mutators**

Accessor and mutator functions are used throughout the solution to provide controlled access to an object’s internal state. For example, the Change class provides accessors such as getName() and getCoinAmt() to retrieve the person’s name and change amount, respectively, and mutator functions such as setCoinAmt() to change the change amount. Similarly, the CoinCountNode class provides accessor and mutator functions to access and update the count, amount, and references to the next node in the linked list. Classes enforce information hiding by encapsulating access to the internal state through accessor and mutator functions and provide a clean separation between interface and implementation details.

**Helper Methods**

Helper methods are used in various classes to encapsulate everyday tasks and promote code reuse. For example, the CoinNodeList class provides helper methods such as length() and toString() to calculate the size of a linked list and convert it to a string representation. Similarly, the CoinDenominationCountry class can include helper methods for retrieving the value of a particular coin or validating the coin’s configuration. By encapsulating reusable logic in helper methods, classes follow the DRY (Don’t Repeat Yourself) principle, improving maintainability and readability by reducing code duplication and promoting modular design.

# **6. Limitations**

Due to project limitations, the program allows users to enter symbols when specifying a person’s name. This accommodation for symbols in the input results from constraints within the project scope that may limit the implementation of stricter input validation rules. However, this tolerance for including symbols in name can pose potential challenges, as symbols may not conform to expected data formats or processing requirements.

A screenshot of a computer

Description automatically generated

Another limitation is when the user inputs the value 10,000,000,000,000 for an integer input in the Java program, and an error occurs because it exceeds the maximum value that the int data type can represent. In Java, int can contain up to 2,147,483,647, which is less than 10,000,000,000,000. Any attempt to assign or parse a value above this maximum results in an overflow situation in which the value jumps to the negative minimum of int (2,147,483,648) and continues. This behavior is dictated by the Java Language Specification, which defines the scope and representation of integer types but does not provide any built-in mechanisms to handle or prevent overflow in arithmetic operations.

A screenshot of a computer program

Description automatically generatedA screen shot of a computer screen

Description automatically generated

# **7. Testing**

**Test table**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Test ID** | **Test description/justification – what is the test for and why this particular test** | **Actual data for this test** | **Expected output** | **Actual desk check result when desk check is carried out** | **Desk check outcome - Pass/Fail** |
| 1 | User-defined name validation: This test verifies the program’s ability to validate custom names to ensure they contain only alphabetic characters. This is very important to maintain data integrity and ensure correct processing. The test case used is “Tom123”, which contains alphanumeric characters, which violates the expected format. | “Tom123” | Error message and prompt for another input | Program process as usual | Pass |
| 2 | User-defined name validation: This test checks whether the program correctly identifies custom names that contain only numbers. Validation should reject such names to conform to the expected format of alphabetic names. Since the input data “123” has only numbers, an error message is displayed, and a new input is requested. | 123 | Error message and prompt for another input | Display an error message and prompt for another input | Pass |
| 3 | Coin Value Validation: This test ensures that the program correctly validates the coin value entered by the user and only allows multiples of 5. The data “12300” represents the value of a coin that is not a multiple of 5 and requires the program to ask if the user wants to add another person. | 12300 | The program proceeds and asks if the user wants to add another person | The program accepts user input and asks if the user wants to add another person | Pass |
| 4 | Coin values validation: Similar to the previous test, this case checks how the program responds when an invalid coin value containing a numeric value not multiple of 5 | 123456 | The program displays an error message and prompts the user for another input. | Program proceeds as expected | Pass |
| 5 | Coin values validation: This test is similar to the previous test. This checks for the alphabet in the input to test how the program will respond. | “Foobar” | The program displays an error message and prompts the user for another input. | Program proceeds as expected | Pass |
| 6 | Case-insensitivity of binary answer. This test verifies that the program correctly handles binary responses in a case-insensitive manner. | “y” | The program accepts user-choice input prompts for a new person’s name | Program proceeds as expected | Pass |
| 7 | Case-insensitivity of binary answer: Similar to the previous test, this case examines the program’s response to uppercase binary reactions. | “Y” | The program accepts the user's choice input and prompts for a new person’s name. | The program ran as expected | Pass |
| 8 | Case-insensitivity of binary answer: This test checks whether the program correctly handles lowercase binary responses for menu options. | “n” | The program should accept the user's choice input and display the main menu options. | The program ran as expected | Pass |
| 9 | Case-insensitivity of binary answer: Similar to the previous test, this case examines the program’s response to uppercase input for menu options | “N” | The program should accept the user's choice and display the main menu options. | The program ran as expected | Pass |
| 10 | Acceptance of provided responses: This test checks whether the program only accepts predefined answers and rejects invalid input | 123 | The program should raise an error message and prompt the user for another input. | The program proceeded as expected | Pass |
| 11 | Acceptance of provided responses: Similar to the previous test, this test checks the responses for alphanumerical value. | “Foobar” | The program should raise an error message and prompt the user for another input. | The program proceeded as expected | Pass |
| 12 | Menu selection validation: This test ensures that the program validates menu selection and only allows options that appear in the menu | 3 | The program should accept the input and display an option menu for input | The program proceeded as expected | Pass |
| 13 | Menu selection validation: Similar to the previous. Test, this case examines the program’s response to invalid menu input, such as non-numeric characters. | “FooBar” | The program should raise an error message and prompt the user for another input. | The program accepts the input | Fail |
| 14 | Case-sensitivity of name entry: This test checks whether the program is case-sensitive when entering names to indicate changes | “jasmine” | The program raises an error message and prompts for another input | The program runs as expected | Pass |
| 15 | Case-sensitivity of name entry: Similar to the previous test, this case tests how the program will respond to names with correct capitalized letters. | “Jasmine” | The program should accept input | The program ran as expected | Pass |
| 16 | Access to main menu functionality: This test verifies that users can access all main menu features 1through 6 | 1, 2, 3, 4, 5, 6 | The program should accept all individual inputs and show appropriate menu | The program ran as expected | Pass |
| 17 | Total change summing: This test checks whether the program can accurately sum up the total change give the same name. | Name: Tom  Coin value: 250, 500 | The program should sum up the total change provided by individuals with same name | The program ran as expected | Pass |

**Result of program testing**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Test ID** | **Test description/justification – what is the test for and why is this particular test?** | **Actual data for**  **this test** | **Expected an output** | **Actual program output when test is carried out** | **Test run outcome –**  **Pass/Fail** |
| 1 | A user-defined name containing alphanumeric. The program should be able to detect if an individual’s name mentioned under user self-defined data has only alphabets. This ensures that the program is as realistic as possible. | “Tom123” | The program should return an error message and prompt for another input. | The program proceeds as usual.  A screen shot of a computer screen  Description automatically generated | Fail |
| 2 | A user-defined name containing numeric. The program should be able to detect if an individual’s name mentioned under user self-defined data has only alphabets. This is to ensure that the. The program is as realistic as possible. | 123 | The program should return an error message and prompt for another input | The program showed an error message and prompt for another input.  A black background with white text  Description automatically generated | Pass |
| 3 | The program can only take multiples of 5; are any other multiples/alphabet to be rejected to ensure the success of the denominator? | 12300 | The program should proceed and ask if the user wants to add another person. | The program accepted the value and asked if the user wanted to add another person.  A black background with white text  Description automatically generated | Pass |
| 4 | The program can only take multiples of 5; are any other multiples/alphabet to be rejected to ensure the success of the denominator? | 12346 / | The program will show an error message and prompt the user for input again. | The program prompted an error message and requested the user to input the values again.  A black background with white text  Description automatically generated | Pass |
| 5 | The program can only take multiple of 5 in integer format. Other inputs are rejected. | ‘Foobar’ | The program should show an error message and prompt the user for another input. | The program proceeded as expected, raising an error message when invalid input was provided.  A black background with white text  Description automatically generated | Pass |
| 6 | The program should not be case-sensitive when prompting the user for a binary answer. | ‘y’ | The program should proceed and ask for the new person's name | The program proceeds and asks for the new person's name. | Pass |
| 7 | The program should not be case-sensitive when prompting the user for a binary answer. | ‘Y’ | The program should proceed and ask for the new person's name | The program proceeds and asks for the new person’s name. | Pass |
| 8 | The program should not be case-sensitive when prompting the user for a binary answer. | ‘n’ | The program should proceed and display the main menu | The program proceeds and displays the main menu.  A screen shot of a computer screen  Description automatically generated | Pass |
| 9 | The program should not be case-sensitive when prompting the user for a binary answer. | ‘N’ | The program should proceed and display the main menu | The program proceeds and displays the main menu.  A screen shot of a computer  Description automatically generated | Pass |
| 10 | The program should only accept the provided responses. | 123 / | The program will show an error message and prompt the user for input again. | The program showed an error and prompted the user to input again.  A black background with white text  Description automatically generated | Pass |
| 11 | Like the previous test, the program should only accept predefined responses. | ‘Foobar’ | The program will show an error message and prompt the user for another input. |  | Pass |
| 12 | The program should only allow selections displayed on the menu; any additional selections should raise an error. | 3 / | The program raised a mistake and should request a new input | A screen shot of a computer error  Description automatically generated | Pass |
| 13 | The program should only allow numeric input for the menu. Any additional inputs should raise an error. | ‘Foobar’ | The program crashes with the “InputMismatchException” error. | A screen shot of a computer program  Description automatically generated | Fail |
| 14 | The program should display the following menu when the previous menu selection is valid. This is tested to ensure unintentional inputs are dealt with? | 2 | The program displayed the following menu. | The program displayed the main menu after the previous correct selection.  A screen shot of a computer  Description automatically generated | Pass |

# Main Menu

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Test ID** | **Test description/justification – what is the test for, and why is this particular test?** | **Actual data for**  **this test** | **Expected output** | **Actual program output when test is carried out** | **Test run outcome –**  **Pass/Fail** |
| 15 | The program is case-sensitive when entering a name to display change. This is tested to ensure case sensitivity, as mentioned. | ‘jasmine’ | The program should display an error and return to the main menu | The program displayed an error and proceeded to show the main menu again.  A screenshot of a computer screen  Description automatically generated | Pass |
| 16 | The program is case-sensitive when entering a name to display change. This is tested to ensure case sensitivity, as mentioned. | ‘Jasmine’ | The program should display the mentioned user number of cents and its denomination before returning to the main menu again. | The program displayed the mentioned user number of cents and its denomination before showing the main menu again.  A screenshot of a computer program  Description automatically generated  OR  The program displayed the mentioned user number of cents and its denomination.  is A screen shot of a computer  Description automatically generated | Pass |
| 17 | Users can access all main menu functionality ranging from 1 to 6 | 1, 2, 3, 4, 5, 6 | The program should show the appropriate menu function | The program showed the correct menu for the option chosen.  A screen shot of a computer screen  Description automatically generated  Option 1  6A screen shot of a computer  Description automatically generated  Option 2  A screen shot of a computer screen  Description automatically generated  Option 3  A screenshot of a computer error  Description automatically generated  Option 4  A screen shot of a computer screen  Description automatically generated  Option 5  A screen shot of a computer  Description automatically generated  Option 6 | Pass |
| 18 | The program should be able to sum up the total change given the same name. This ensures that individuals can sum up their total coins if they did not enter the correct amount initially. | Name: Tom  1st Coin Value – 250  2nd Coin Value - 500 | The program should sum the total of changes provided by individuals with the same name. | The program displayed the correct sum of the total coins of the same name.  A screenshot of a computer  Description automatically generated | Pass |