1. IDENTIFICATION OF APPLICATION:

1.1. The application consists of 4 classes: `BankAccounts, BankAccount, Customer, Transaction.`

1.2. Version 1.0 of the application and all its classes and functions were written by *Manfred Kerber* [m.kerber@cs.bham.ac.uk] on 20th October 2015.

2. DESCRIPTION OF TESTING PROCEDURE:

2.1. Due to the lack of a user interface the testing can be limited entirely to the use of JUnit testing cases (JUnit version 4.11).

2.2. **Scope of current tests:**

- 2.2.1. Coordinated functions of critical units;
- 2.2.2. Methods providing program's functionality;
- 2.2.3. Adequate exception handling, error messages.

2.3. **Out of current tests' scope:**

- 2.3.1. Password protection mechanism which is to be implemented in future versions of the application (virtually non-existent in current version: plain-text);
- 2.3.2. Web environments, cross-platform portability & different system architectures (written for the JRE 8).

2.4. **Test items:**

- 2.4.1. Various methods and functions of the application are grouped in 2 major units *(see **Figure 1.**)*:
  - 2.4.1.1. **Access Unit:** manages the access and modification of previously recorded data;
  - 2.4.1.2. **Create Unit:** manages the process of registering a new account.
- 2.4.2. The tests' primary focus is the overall coherence, and individual functionality and security of the program *(see **Table 1.**)*.
- 2.4.3. Test Item pass/fail criteria:
  - 2.4.3.1. BankAccount functions must at all times return correct numeric values when relevant to bank balance: incorrect calculations are naturally unacceptable;
  - 2.4.3.2. Error messages & exceptions must be thrown when incorrect values are entered in order to avoid incorrect data registration and further misbehaviour that is unaccounted for.

3. SOFTWARE RISK ISSUES:

3.1. The risk level of each individual element (high, medium, low) is evaluated based on how likely it is that the unit in question jeopardizes the integrity and security of the application *(see **Table 1.**)*;

3.2. Password-protection systems (password-hashing) are yet to be developed, considering the importance of privacy when dealing with applications pertaining to financial security;

3.3. Due to the lack of a GUI, the application must to be sufficiently documented in order for the average user to be easy-to-use, exceptions and error messages must be adequate and precise.

***Figure 1.* Function Units**

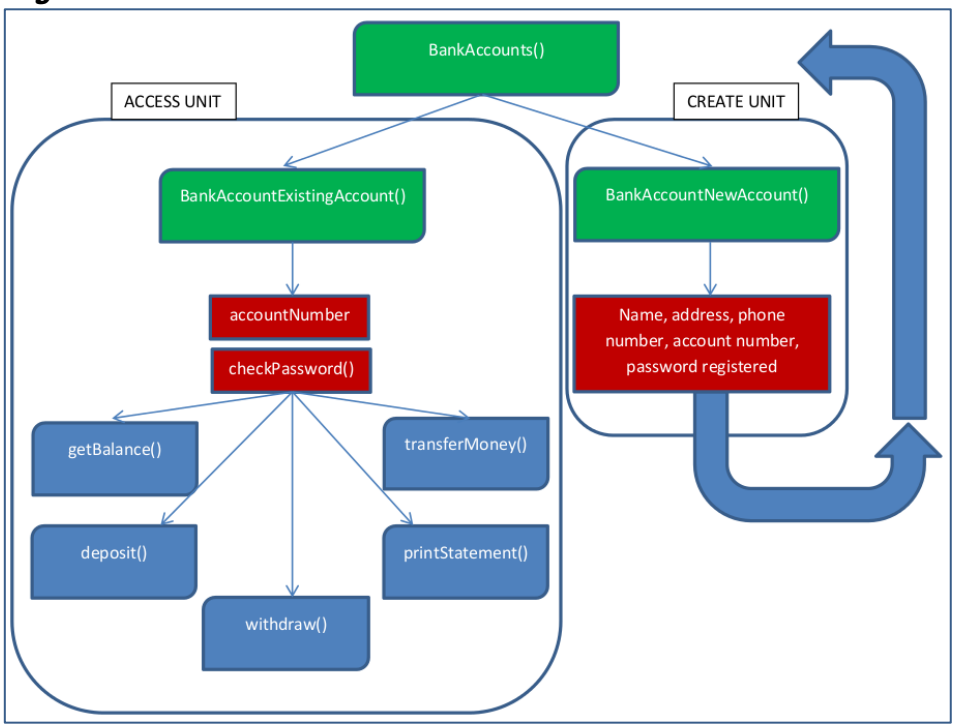

Table 1. Testing of Critical Units & Mechanisms

1. ACCESS UNIT			
RISK PRIORITY	**NAME OF FUNCTION**	**EXPECTED FUNCTIONALITY**	**HOW TO TEST in JUnit**
HIGH RISK	`Bank-Account-Existing-Account method`	Looks up stored account number's data, returns error message if not found	• Set-up 2 `BankAccount` objects and test if their data is accessible through the method • Attempt to access an inexistent account to prompt an 'account not found' error message
HIGH RISK	`check-Password method`	Checks if password entered by the user matches the password stored for the account number, returns error message for invalid characters	• Create test object (account & password) • Test if correct account is accessed upon correct password insertion • Test if any account is accessed upon incorrect password insertion • Attempt to set a password using invalid characters to prompt an 'invalid character' error message
LOW RISK	`print-Balance method`	Prints current bank balance using the correct formatting & currency symbol	• Set-up a `BankAccount` object and deposit some amount in it • Call the `printBalance` method to see if the output is correct
MEDIUM RISK	`deposit method`	Records deposited amount & adds it to the balance	• Set-up a `BankAccount` object and deposit some amount • Test if correct amount has been registered in conjunction with the `printBalance` method
MEDIUM RISK	`transfer-Money method`	Withdraws money from one account (decreases the balance) and records it as an in-payment to another account, returns error messages if: the bank account numbers are incorrect and if the requested funds are not available	• Set-up 2 `BankAccount` objects and deposit some amount in one of them • Test if method correctly allows the transfer into the second account using the `printBalance` method on the second account
HIGH RISK	`withdraw method`	Allows the recording of the withdrawal of a certain sum, if the bank balance has the sufficient funds, returns error message otherwise	• Set-up a `BankAccount` object and deposit some amount • Withdraw some amount using the `withdraw` method • Test if method has correctly adjusted the new balance using the `printBalance` method
2. CREATE UNIT			
RISK PRIORITY	**NAME OF FUNCTION**	**EXPECTED FUNCTIONALITY**	**HOW TO TEST in JUnit**
MEDIUM RISK	`Bank-Account-NewAccount Method` **& pertaining setters & getters**	Creates a new bank account entry and assigns it an account number, returns error message for invalid characters	• Use to create a `BankAccount` object & try to access it by calling the assigned account number • Attempt to create an account object using invalid characters to prompt an 'invalid character' error message