Case study for demonstration of TDD in Java

The following is the specification of the Banking Application case study that will be demonstrated as part of applying TDD for creating and designing the application code.

The user stories to be implemented are as follows.

1. As a user I should be able to manage my account in the bank by deposit and withdraw functionalities and should be able to verify the account balance updated by deposit and withdraw transactions.
2. The Bank account will have three different types
    a. **SavingAccoun**t
    b. LoanAccount
    c. CurrentAccount
3. In each of the above account type behaviour for deposit and withdraw will vary as below
    a. **SavingAccount**: Deposit will add amount to balance and withdraw will subtract the amount from balance. The negative amount passed in deposit and withdraw should have no impact on balance. The account can have an initial balance.
    b. **LoanAccount**: Deposit will subtract amount from loanBalance amount if valid loan present and withdraw will allow only if loanBalance is zero and will subtract the amount from balance. The negative amount passed in deposit and withdraw should have no impact on balance. The account can have an initial balance.
    c. **CurrentAccount** : Deposit will add amount to balance and withdraw will allow only if balance is greater than 1000 zero and will subtract the amount from balance. The negative amount passed in deposit and withdraw should have no impact on balance. The account can have an initial balance.

4. **Development setup**: Windows 7/10 with JDK1.8,Eclipse-Jee-Photon,
5. **Steps for assignment**
    1. Define the test classes in test package with test methods to exercise the tests for SavingAccount functional behaviour.
    2. Create domain classes in com.bank package.
    3. Run the tests and add required domain code for the tests to succeed
    4. Repeat the steps for other two account types

6. **During the TDD cycle**
    1. Define the test
    2. Run the test.
    3. let it fail first time..The Red bar will be shown
    4. Apply the required minimum implementation code to make the test succeed i.e. Green bar

5. Fake the implementation
6. Write obvious code for implementation
7. Apply generalized approach when the code becomes duplicate and dependent(i.e. Triangulation)
8. Run the test to see the green bar.
9. Run all the tests to check the success of all the tests.
10. Add more test methods to test more behaviour of account
11. Refactor the code to remove duplicate code and easier maintenance.
12. Run all the tests again to verify the success of refactoring.

7. Repeat the same process for other two account types

8. Apply Code Refactoring

1. Remove the common duplicate code across different types of account and abstract it into a top level abstraction of Account.
2. Run all the tests and make all of them succeed.

9. Separate the process of object creation and usage.

1. A factory should return the account type objects to the program instead of creating them in the programs.
2. Define test class to test the object creation from a factory class by specifying type of account
3. Define Account enum type to specify the range of account types.
4. Define factory class with required functions.
5. Test the factory with three different types of accounts for success.
6. Run all the test methods for success.

******