Bank Work Test Assignment for JavaEE Developers

rev. 1.2.0 - 2013-10-31

1 Introduction

This document describes the requirements for completing a basic Java development assignment. The purpose of the assignment is to get an indication of a developer's ability to design and build solutions using modern JavaSE/EE technology.

This document together with the source code informs you what the assignment is about. There are some strict guidelines on how the work needs to be performed. These guidelines and instructions are used to ensure consistency and make work tests comparable. You should read these instructions carefully before you begin work on the solution.

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC2119.

1.1 Contents

The distribution archive consists of the following:

- This document
- Source code that defines the contracts you need to implement
- Unit tests that demonstrate key functionality
- Complete project structure and build file

1.2 Background

The IT department recently had an undergraduate student in for work experience during the last few months. This student was tasked to implement a basic double-entry bookkeeping banking service. Unfortunately time ran out and only the API design for the project was completed.

The IT department now wishes to extend this work into a functional application or service. All functionality has been defined in the high-level business interfaces and test classes.

Your mission is to provide a working solution by implementing these interfaces.

1.3 Project Status

The banking service already contains the following interfaces that MUST be implemented in order to complete the project.

- **AccountService** – Business service for creating monetary accounts and query for account balance.

- **TransferService** – Business service for executing multi-legged monetary transactions towards accounts.

- **BankFactory** - Factory interface for providing instances of the business services described above.

The test classes “BankUnitTest” and "BankFunctionalTest" demonstrate the functionality of the interfaces. Notice that “BankFunctionalTest” is not actually runnable until you have implemented some solution.

2 What you must do

This section lists the functionality that MUST be implemented to successfully complete the assignment. First, some words about the double-entry accounting principle:

"Double-entry bookkeeping involves making at least two entries or legs for every transaction. A debit in one account and a corresponding credit in another account. The sum of all debits should always equal the sum of all credits, providing a simple way to check for errors."

Key requirements

- **Ability to create monetary accounts with an initial balance**
  A client is allowed to create its own accounts with an initial balance (for simplicity).
- **Ability to transfer money between accounts**
  It must be possible to move money between accounts using multi-legged, balanced transactions following the double-entry accounting principle.
- **Durable storage of accounts and transaction history**
  All accounts and financial transactions must be stored in a relational database persistent store for later retrieval.

Definitions

- **Client -** A client is an application using the service, in this case the unit tests
- **Account** - A system record with an arbitrary client-defined reference and a Money balance
- **Money -** A monetary unit consisting of an amount and ISO-4217 currency code
- **Transaction -** A financial transaction between at least two different accounts
- **Transaction Leg** - An element of a transaction defining a credit or debit towards a single account
- **Transaction type -** A looosely defined type or context for a financial transaction

Business Rules

- An account MUST NOT be overdrawn, i.e. have a negative balance.

- A monetary transaction MAY support multiple currencies as long as the total balance for the transaction legs with the same currency is zero.

- The concepts of *debit* and *credit* are simplified by specifying that monetary transactions towards an account can have either a positive or negative value.

3 How you must do it

3.1 Modifying Source Code

It is NOT allowed to modify existing Java source code including the test classes. You MAY edit the build files to make necessary adjustments for your solution.

3.2 Implementation

All methods in the main business services described above MUST be implemented.

A working implementation of the **BankFactory** interface MUST be named and located in **"com.unibet.worktest.bank.BankFactoryImpl"**.

All other source code you write MUST be placed in a different package than "com.unibet.worktest.bank".

3.3 Documentation

The code you write SHOULD be clear, consistent and self-documenting. A brief but clear description of the solution design, technology choices, limitations, possible deployment instructions and so forth MUST be included. UML diagrams or other drawings are OPTIONAL. All documentation that is not in code MUST be in either plain text or PDF format.

3.4 Currency Conversion

No support for currency conversion is required.

3.5 Presentation Layer

No presentation layer is required for the system at the moment.

3.6 Remoting

No remoting API is required for the system at the moment.

3.7 Non-functional Requirements

- The solution MUST be able to handle *concurrent requests* in a safe way
- The solution MUST use a RDBMS for persistent storage

3.8 Platform and technology

You are free to use any implementation technology of choice as long as its based on Java platform JDK 1.7 or later. Use of 3rd party frameworks and products is allowed and encouraged as long as they are open-source.

If you prefer to use a JavaEE container the acceptable choices are Tomcat, Jetty or Glassfish. It is not important which version of the containers you choose. If the solution requires a container, then don't forget to provide simple deployment and running instructions.

Acceptable RDBMS choices are PostgreSQL, MySQL or H2 in embedded mode. Versions are not important but you must describe which one you have used.

4 Building

The solution must be built with Maven 3.x.

http://maven.apache.org/download.html

A skeletal pom.xml file is already provided. The dependencies listed in the pom.xml MAY be used. Any additional dependencies that are not available in official public Maven repositories MUST use *system* scope and be included in a *lib* directory.

The solution MUST be possible to build and test using the "mvn test" goal.

5 Deliverables

The solution deliverable must consist of a single ZIP file or TAR.GZ. The archive MUST contain the original directory hierarchy along with added files or directories needed for the implementation.

In all:

- Source code for new classes and any modified build files, SQL script files and so forth
- A brief solution text file or PDF describing to the reviewer the following information:
  - Your major design choices and the reason for those choices
  - The exact JDK and JRE versions used
  - RDBMS product and version used
  - Any application server product and version used
  - How to deploy the solution in case it needs a container
  - The location of any setup scripts and how to execute them

6 Evaluation Process

This section decribes how your solution will be evaluated. First we run the code ensuring that all requirements are fulfilled. If that is satisfactory, then we investigate the design and implementation details of the solution. Mainly the following areas:

- **General**
  - Overall design choices, data model and patterns used

- Technology choices and motivations for those choices
- Functional correctness
- Non-functional correctness, i.e. robustness and transactional safety

- **Documentation**
  - Javadoc source documentation and comments
  - Design documentation and motivations
- **Code Quality**
  - Error handling
  - Testability
  - Use of coding standards, code clarity and readability

7 Final Words

There is no time limit on completing the assignment. Good luck.