INFO213 Course Project
Bank Teller's Simple Banking Application Development

1. Introduction

Several examples in lectures and tutorials in INFO213 have been borrowing classes, features, and data from the domain of personal banking. Needless to say, these examples offer a drastically simplified glimpse into the personal banking knowledge/practice domain — we are putting aside the myriad of highly-specialised fundamental and intricacies to be found in this domain. However, the basic concepts from the personal banking domain lend themselves as an appropriate tool for learning Object-Oriented Systems Development.

The properties of personal banking domain objects within the `SimpleBankModel` schema, relationships between objects, features of these objects, combined with the basic Graphic User Interface (GUI) features and functionality in the `SimpleBankView` schema provide us with a rich learning playground. The JADE Development Environment in one application offers features for testing our ideas, experimenting, and letting our creative juices flowing, while learning the powerful functionality provided by the JADE Platform with its built-in object persistence, class/data definition and management tools, and the multitude of the development tools in the JADE IDE.

The theme of the course project follows the personal banking domain line of learning exploration — we are already familiar with the basic parameters of this space, which saves us a bit of time, if we were to choose a different target domain for the course project. And so, in the course project you are required to develop a prototype of a desktop application which would be appropriate for the use case of a bank teller, where the teller could perform such actions as adding/editing customer details, creating/editing bank account details, along with viewing and searching transactions, or processing deposits.

The design of the GUI for the bank teller application has a lot of scope for creativity, and while desktop GUI applications are forfeiting their ground to Web-based applications, the similarities in the overall design considerations are appropriate for our learning context. It goes without saying that application interconnectedness sits at the core of any modern system — data processing/management is the focus of every information system, where the functionality of these systems is distributed across networked application and database servers. One feature of the bank teller application to be developed in this course project utilizes network connectivity features to send data for remote processing/storage on an entirely different system, which potentially may be a seen as a basic example of business-to-business (B2B), or even business-to-government (B2G) data sharing/communication.

The rest of this document provides the specific details of the tasks to be undertaken during this project, tips for getting started with the project, project submission requirement, and last, but not least, the project marking schedule.

2. Task Specifications

The Bank Teller's Simple Banking Application solution has the following requirements, listed below as use cases. Naturally, we are leaving many other features desirable in a bank teller application beyond the scope of this project to limit the design and workload effort/requirements.

2.1. Basic GUI Features/Use Cases

The application should offer appropriate GUI Multiple Document Interface (MDI)-based interface features/functionality for:

- Adding/editing customer details, searching customers by name or number.
- Adding/editing accounts for customers, searching accounts by number.
- Searching transactions and making (cash) deposits into customers' accounts.

2.2. XML Data Exchange Use Case

In addition, the application should implement a feature to send details of a set of individual account transactions to an unnamed (in this instance) government organization, say, for the purposes of investigating suspected tax evasion or money laundering or other kinds of financial fraud.

This feature must include seamless conversion of customer/account/transaction data to XML format with subsequent transfer of the XML data to a very basic Representational State Transfer (REST, RESTful) Application Programme Interface (API) endpoint. The functionality on the API side is limited to data validation to ensure that data is submitter in correct format. The format of the XML data is demonstrated in the files/resources supplied along with this course project specifications.

The use case must be implemented in a way to allow a bank teller to select a customer account/transaction to be shared with this B2G interface, starting the process via appropriate GUI menu (sending the data) and demonstrating the API response in an appropriate manner in the GUI.

2.3. Error Handling/Logging

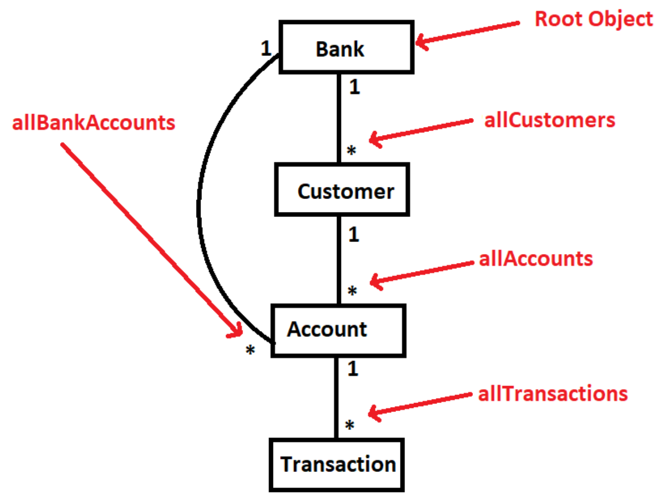
Where applicable, the application should implement appropriate, yet basic, error handling — for this purpose it is advisable to implement a small set (three-to-five) user exceptions along with appropriate handler code and logging to disk file. Error handling is essential, for example for data import/export features, and GUI/user error prevention.

3. Getting Started

It is recommended that you start with the versions of the `SimpleBankModel` and `SimpleBankView` schema posted as solutions to Tutorial W07, although you may feel free to delete any of the existing methods, forms, etc. Then, the scope of work for this course project may be divided into three categories:

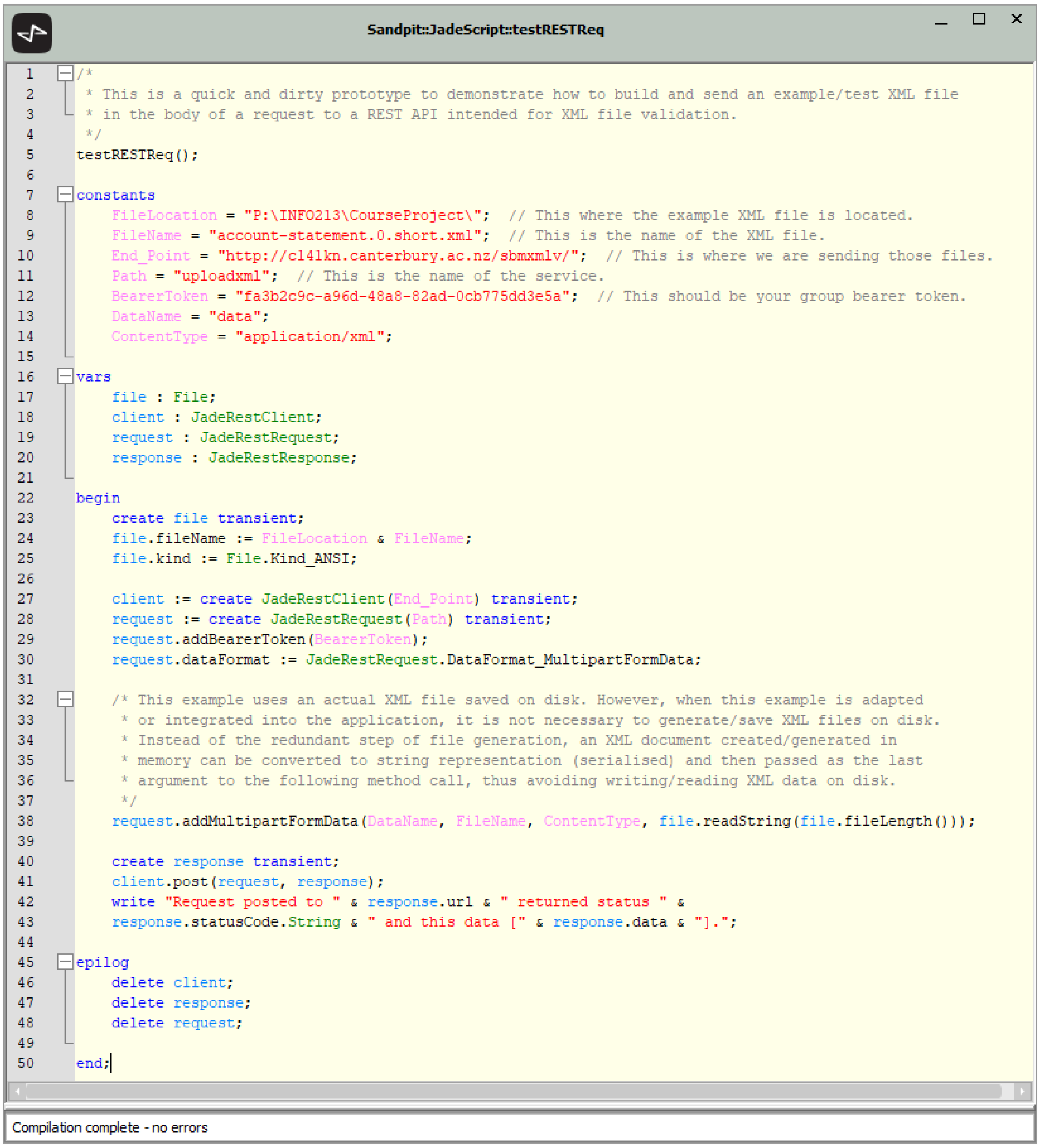
1. Creation of the appropriate object model to match the requirements of the bank teller application,
2. Design and development of the GUI to implement the use cases for operations on customer, account, and transaction data,
3. Finally, the work required to implement the XML data exchange features.

To get started with the first category, recall the comprehensive collections structure as demonstrated in lectures — it is recommended that you take this model as the foundation of the object model in the application:



Then, the second category will require a lot of brainstorming, discussions, prototyping, and trial and error. This is a non-trivial task — there are a lot of features requiring several different GUI controls, appropriate to the task, which need to be combined and arranged with good GUI design principles and heuristics in mind.

And then, to get you started with the third aspect of the functionality, the XML data exchange use case, consider the following example:



```
/*
 * This is a quick and dirty prototype to demonstrate how to build and send an example/test XML file
 * in the body of a request to a REST API intended for XML file validation.
 */
testRESTReq();

constants
    FileLocation = "P:\INFO213\CourseProject\";  // This where the example XML file is located.
    FileName = "account-statement.0.short.xml";  // This is the name of the XML file.
    End_Point = "http://cl41kn.canterbury.ac.nz/sbmxmlv/";  // This is where we are sending those files.
    Path = "uploadxml";  // This is the name of the service.
    BearerToken = "fa3b2c9c-a96d-48a8-82ad-0cb775dd3e5a";  // This should be your group bearer token.
    DataName = "data";
    ContentType = "application/xml";

vars
    file : File;
    client : JadeRestClient;
    request : JadeRestRequest;
    response : JadeRestResponse;

begin
    create file transient;
    file.fileName := FileLocation & FileName;
    file.kind := File.Kind_ANSI;

    client := create JadeRestClient(End_Point) transient;
    request := create JadeRestRequest(Path) transient;
    request.addBearerToken(BearerToken);
    request.dataFormat := JadeRestRequest.DataFormat_MultipartFormData;

    /* This example uses an actual XML file saved on disk. However, when this example is adapted
     * or integrated into the application, it is not necessary to generate/save XML files on disk.
     * Instead of the redundant step of file generation, an XML document created/generated in
     * memory can be converted to string representation (serialised) and then passed as the last
     * argument to the following method call, thus avoiding writing/reading XML data on disk.
     */
    request.addMultipartFormData(DataName, FileName, ContentType, file.readString(file.fileLength()));

    create response transient;
    client.post(request, response);
    write "Request posted to " & response.url & " returned status " &
    response.statusCode.String & " and this data [" & response.data & "].";

epilog
    delete client;
    delete response;
    delete request;

end;
```

```
Compilation complete - no errors
```

4. Submission Requirements

The project is to be submitted as a single .zip file via UC LEARN | INFO213 | Assessment 3: Course Project.

1. The submission must include a report/documentation in MS Word or PDF format. Include either individual or group assignment coversheet (whichever is applicable) — those are available on UC LEARN | UC Business School Students | Cover Sheets page (download a coversheet, fill it out, scan it

(*yes, scan it rather than take a photo!*), and include it as the first page of the report. There are no min/max report page count — instead the report must be complete to outline for the marker the following aspects:

a) The scope of work/features completed, along with UML diagrams and screenshots, where applicable. This should list the implemented features, along with your design choices.

b) Walk-through instructions to guide the marker to test the implementation of the required feature and, especially to show case your design finesse and highlight what you feel requires highlighting in your solution/submission. Include a description of the error handling with a walk through to verify the implementation.

c) Sources of information, tools, etc., which were used in your solution — the Internet (and the rest of the world is your oyster), but it is essential you are specific about borrowing/using designs/ideas that are not your own. For example, if you used some screenshots/design ideas you've found elsewhere to get yourself started — mention the source; similarly, if you used ChatGPT or any of its friends to generate synthetic test data — state what it was.

2. Code — the schemas for your solution. Keep in mind that the schemas will be loaded by the markers of the project — make sure they can be loaded without any issues and the code runs as intended in the first place.
3. Data — if your application requires any additional data, make sure it is included in the submission.

5. Marking Schedule

Report/Documentation **30 Marks**

Simplicity, consistency, presentation quality, use of appropriate diagrams, screenshots, list of sources, etc.

Application GUI and Basic GUI Use Cases Implementation **40 Marks**

Clarity and ease of the GUI implementing the basic use cases, automated collection membership and inverse definitions.

Error Handling **10 Marks**

Basic error handling (to catch errors generated by user input, incomplete data import/export), error logging.

XML Data Exchange **10 Marks**

Generation of XML data files in the correct format, data exchange using REST API, results display in the GUI.

XML Data Import Implementation Using JADE XML Parser (Advanced) **10 Marks**

Loading XML data files subclassing `JadeXMLParser` to import/create objects for `Customer`, `Account` & `Transaction` classes/subclasses.

6. Further Questions & Clarifications

There may be a few things you'd want to clarify — please, make use of our Discussion Forum!

Don't be shy to go ahead and ask questions even if you think they are naïve questions — your classmates with thank you for being brave. You never know what sort of tips and trick may return to you in response to your questions. And then, don't be jump in and answer questions too — it is highly encouraged (and encouraging) to learn together — we only need to be careful not to share complete solutions/code.

IMPORTANT: *The crucial part is not to leave your questions (actually, doing your work and discovering the gaps that need to be filled by asking questions) till the 11th hour. On the teaching team end the common occurrence is to be flooded by questions too close to the deadline when it is not enough time/sense to put effort into answering the questions. It is quite likely that the questions posted too close to the final call will not be answered. Say, the project is due on a Sunday — you probably want to make sure all burning questions are posted at least by Friday morning!*