|  |  |  |  |
| --- | --- | --- | --- |
|  | 3/2/2015 |  | |
| |  |  | | --- | --- | |  |  | |  |  | |  |  | |  |  | | |  | |
| SSE 554 Project 2 Report  *Test Driven Dev., Distributed Version Control, and Extensible Markup Language* | | | |
|  |  | | |  |  |  | | --- | --- | --- | |  |  |  | |  |  |  | |  |  |  | |  |  |  | |  |  |  | |  |  |  | |
|  |  | |  |

|  |  |  |
| --- | --- | --- |
|  | Daryl Ebanks, David May, Josh Deremer, Kei’Shawn Tention |  |

Table of Contents

[Requirements and Testing 2](#_Toc413106346)

[Development Summary 2](#_Toc413106347)

[Requirements (for I/O Class) 2](#_Toc413106348)

[Testing 2](#_Toc413106349)

[Description of Code (with pics!) 3](#_Toc413106350)

[GUI 3](#_Toc413106351)

[XMLIO 7](#_Toc413106352)

[Database 8](#_Toc413106353)

[Distributed Version Control System 8](#_Toc413106354)

[An Picture is worth 1000 words 8](#_Toc413106355)

[Additional Important Topics 9](#_Toc413106356)

[Using a DTD for XML File Validation 9](#_Toc413106357)

[Using a Document Object Model 10](#_Toc413106358)

[Appendix I Full Unit Testing Code 12](#_Toc413106359)

[Appendix II Source Code 17](#_Toc413106360)

# Requirements and Testing

## Development Summary

Once again the TDD process was used to develop portions of this code. A set of tests were developed from the requirements of the classes required to write this application.

## Requirements (for I/O Class)

The requirements were as follows:

* Use XML as a means of storing the information in a file
  + Reading from an XML file and ~~parsing the document for its information~~
    - Building the document tree so it can be used within the program
  + Writing to an XML file with the same
    - Building the document tree so the xml document will be properly formatted.
    - Accepting the document tree

These requirements created the XML I/O Test classes and XML I/O classes.

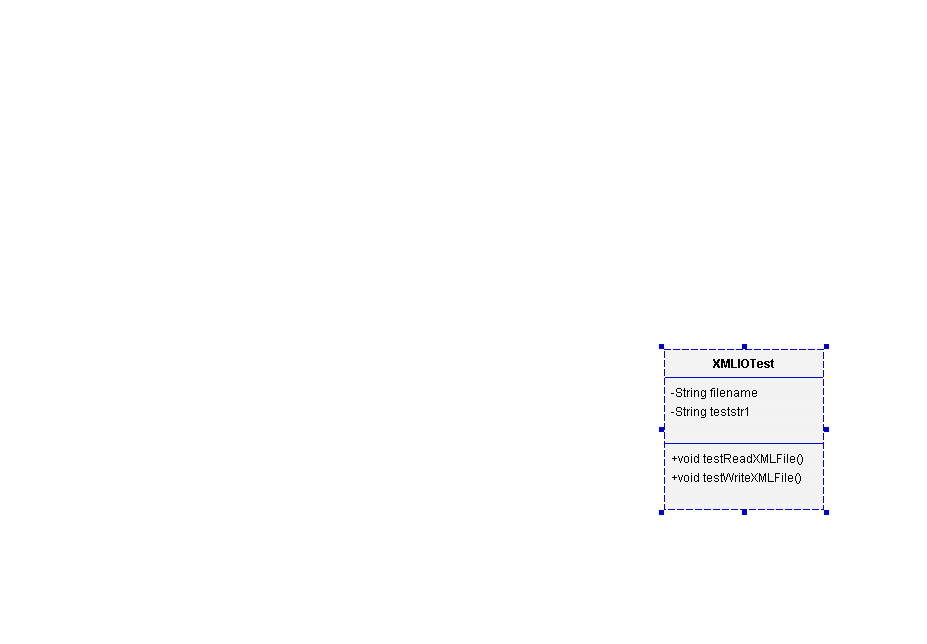
The lines that have a strike were later removed to keep the File I/O about file i/o exclusively.

## Testing

The XML IO Test class: This class tests the readXMLFile and writeXMLFile methods within the XMLIO class. The testing for the accounts had to be slightly changed to accommodate XML and development environment changes.

This is an example of one of these tests:





# Description of Code (with pics!)

The application is a rewrite of the Bank application using XML for data storage and GUI. The account classes and tests are the same as those of the original bank program.

## GUI

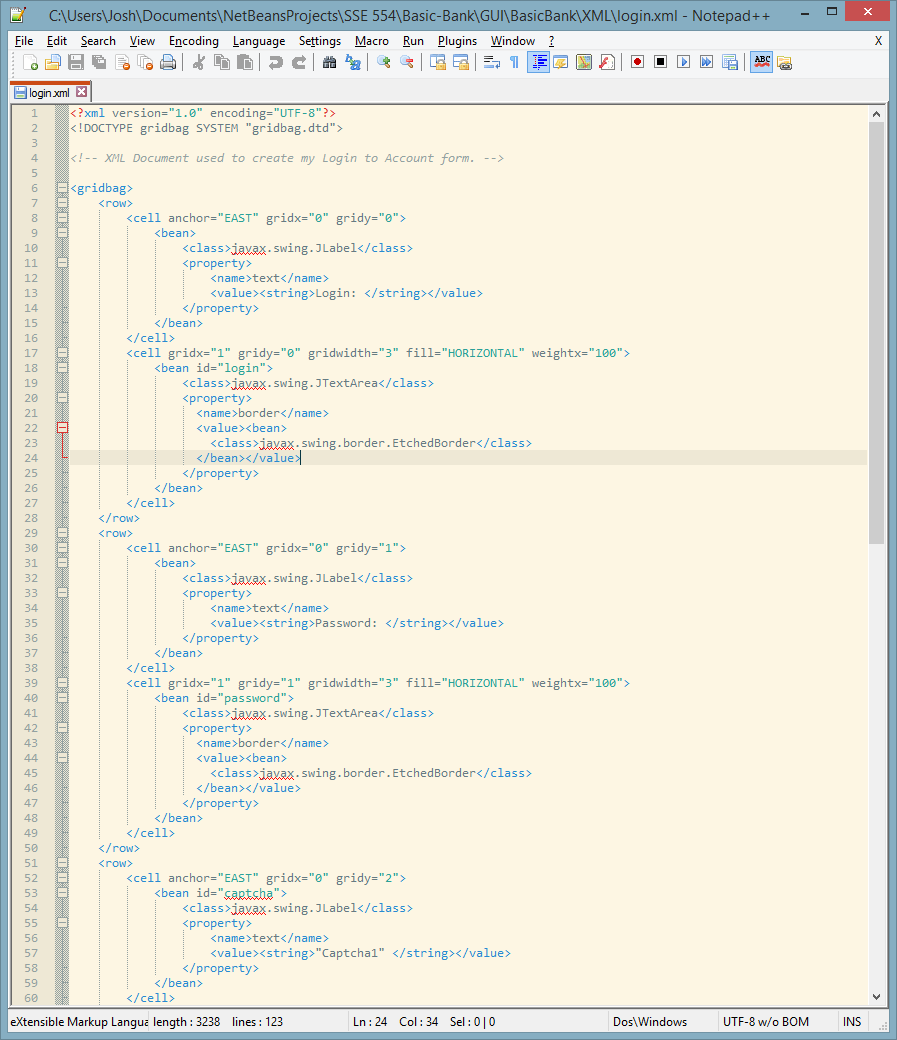
Similar to the first the GUI displays the functions of the account and bank (now basic bank) classes. Some alterations were made to the GUI.

The GUI has been built using XML files formatted using JSwing and the grid bag pane method. This method focuses on building up a GUI in a tabular form. The GUI frame will have a grid of cells across and down that will be populated by different swing components defined in the XML file. The XML file is able to set the properties of each of these swing components as well as determine the component location within the cells and frame resizing.

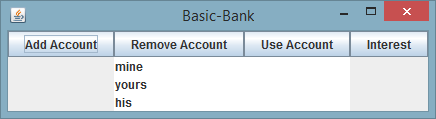
Within the XML file, there are elements for rows, cells and beans. The rows determine vertically where the data following is located, whereas, as one would expect, the cells determine horizontally where it is located. By finding a row, then cell, one can determine approximately where a swing component is located. The bean element type is used to define the swing component that will be in that location as well as a set the properties of said swing component. This method is reproduced for defining the frames of each of the GUIs. A snippet of an XML file used in this project is shown on the next page.

In order to parse through the XML file to build the frame, a DOM was developed in Java similar to the I/O portion. This DOM steps through the XML file creating the tree structure as it goes to store all of the elements. Once it has stored everything, it begins building each of the components of the swing application. This entire process is done using the DocumentBuilderFactory and DocumentBuilder classes provided by the World Wide Web Consortium (W3C). We can use the document created to reference any of the swing components we want.

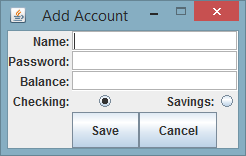
The next step is to turn the DOM into a class that can create a frame. This is done using the GridBagPane class. Within this class, we navigate the nodes of the DOM, pulling out the constraints of each cell and adding the cell as a component to the swing frame. In addition, we navigate through the nodes that are beans, pulling out the components that go inside the cells and setting the property descriptors to the ones defined within the XML file. With this class, we can access any of the swing components to add to them or change them using the get method, or just make the frame visible and see how the components and frame is laid out.



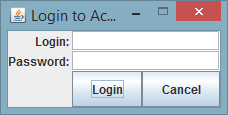
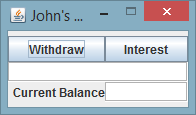
With all of this work done in the background, we were finally ready to make our first frame. The starting frame, BankFrame, contains four buttons and a list of all the accounts currently in the bank. Within this frame, the user of the application can navigate to the add account frame, remove a selected account from the bank, use an account and be transferred to the Login frame, or apply the interest calculation to the accounts. Unfortunately, the grid bag method does not have a way to define listeners for swing components so the code associated with that is done in java. This required created four action listeners, one for each of the buttons, then assigning those action listeners to the buttons. However, everything else was completed by the grid bag method, and so our BankFrame class is complete.



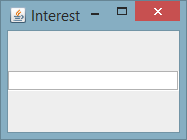
Next, each of the smaller frames was written in XML and then pulled out via Java. The AddAccountFrame class builds a frame that allows the user to enter a name, password and starting balance and choose between a new checking account or savings account. In this frame, the textfields, radio buttons, and buttons all were pulled from the grid bag pane class. The textfields and radio buttons were used to send data to the database while the buttons had action listeners added to them.



Following this, the LoginFrame XML and Java were created. This frame allows a user to login to his or her account if a correct name and password combination is provided. Once again, the textfields and buttons were pulled from the grid bag and referenced or had action listeners appended. If the user logins in correctly, they open a frame that gives the user the chance to make a withdraw or deposit, as well as see their current balance.

Finally, the last frame to be added is the Interest frame. This frame shows a complete list of all accounts and provides details as to the interest operation that was performed on each account through a textbox.



Each of these frames were developed first in XML. Then, the XML was parsed in order to create a DOM tree containing a tree of nodes from the XML file. This DOM tree is traversed, pulling out the location and information for each of the cells that should be in a frame as well as the swing components and properties that populate each of the cells. With this grid bag method, the frame itself becomes populated, but has absolutely no functionality yet. Within each of the frame classes, we must add action listeners for the buttons and pull data that is put into each of the text areas. Once these few lines of code are written, we have a working GUI interface!

## XMLIO

The XMLIO is pretty simple it only has two methods. These methods read and write data to XML files for storage. This portion uses the DOM and passes the document to the database so it can be parsed.

The Document Object Model (DOM) is a method of storing data to make strictly formatted documents simple to parse and generate. This is discussed further in the section entitled Using a Document Object Model.

## Database

The database for the Basic-Bank Application is simple user-defined class that is designed to store the different bank accounts created. The main object that drives the database is an Array List of Account objects. The source code for the Account.java class can be located in the Appendix. The source code for Database.java is located in the Appendix as well. The constructor of the Database.java class has the responsibility of also instantiating the object for the XMLIO.java class and the object for the Document class. The Document object is explained more in the section entitled Using a Document Object Model .

The Database.java class consists of the following methods:

1. addAccount()
2. removeAccount()
3. withdraw()
4. deposit()
5. getBalance()
6. verifyAcct()
7. writeToFile()

There are more methods that have minor roles for the functionality of the Database.java class. The first 5 methods in the list are self-explanatory and intuitive. However, verifyAcct() and writeToFile() need explaining. Account verification is important for users, whether it is a bank teller or a customer, because it allows the right person to authorize transactions to the proper account they logged into. The following page has the block of code that allows for a secure access to an account.

public boolean verifyAcct(Account acct){

boolean temp= false;

for(Account a:db){

if((a.holder.compareTo(acct.holder)==0) && (a.password.compareTo(acct.password)==0))

temp = true;

}

return temp;

}

The writeToFile() method is another important aspect of the Database.java class. It allows all the accounts created to be stored into an XML file using the Document Object Model. This concept was one our team wanted to focus on as a major topic for this project. The block of code is provided below.

public void writeToFile()

{

try

{

DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();

DocumentBuilder builder;

builder = factory.newDocumentBuilder();

Document doc = builder.newDocument();

XMLIO io = new XMLIO();

//root element

Element rootElement = doc.createElement("Bank");

doc.appendChild(rootElement);

for(Account a: db){

//add account elements

Element acct\_Tag = doc.createElement("Account");

Element holder = doc.createElement("Holder");

acct\_Tag.appendChild(holder);

holder.appendChild(doc.createTextNode(a.holder));

Element psswrd = doc.createElement("Password");

acct\_Tag.appendChild(psswrd);

psswrd.appendChild(doc.createTextNode(a.password));

Element balance = doc.createElement("Balance");

acct\_Tag.appendChild(balance);

balance.appendChild(doc.createTextNode(Double.toString(a.balance)));

Element accType = doc.createElement("AccountType");

acct\_Tag.appendChild(accType);

accType.appendChild(doc.createTextNode(a.accType));

rootElement.appendChild(acct\_Tag);

io.WriteXMLFile(doc, new File("XML/Database.xml"));

BufferedReader reader = new BufferedReader(new FileReader("XML/Database.xml"));

String line = null, teststr2="";

while ((line = reader.readLine()) != null) {

teststr2 += line;

}

reader.close();

}

// Files.delete(FileSystems.getDefault().getPath("", "XML/Database.xml"));

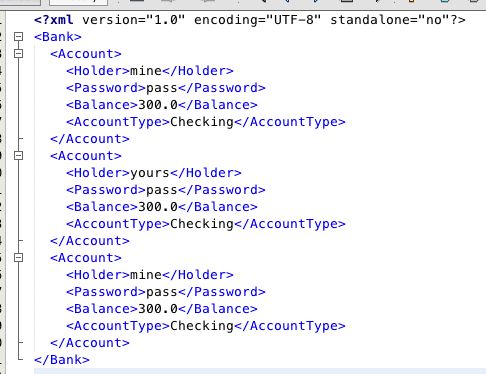
} catch (Exception e) {

e.printStackTrace();

}{}

}

A portion of the Document Object Model was used within the Database.java class. As mentioned before, it was used to store the banks accounts into an XML file. The following figures illustrate the creation of banks accounts, the logging in to a specific account, and the transactions of the accounts. Upon exiting the Basic Bank application, the writeToFile() method is called and the accounts are written to an XML file.



Default accounts of the Basic Bank application.

Main BankFrame. The default accounts can be seen.

Create a New Account.

Click save. Clicking save does not add the account details to the XML file.

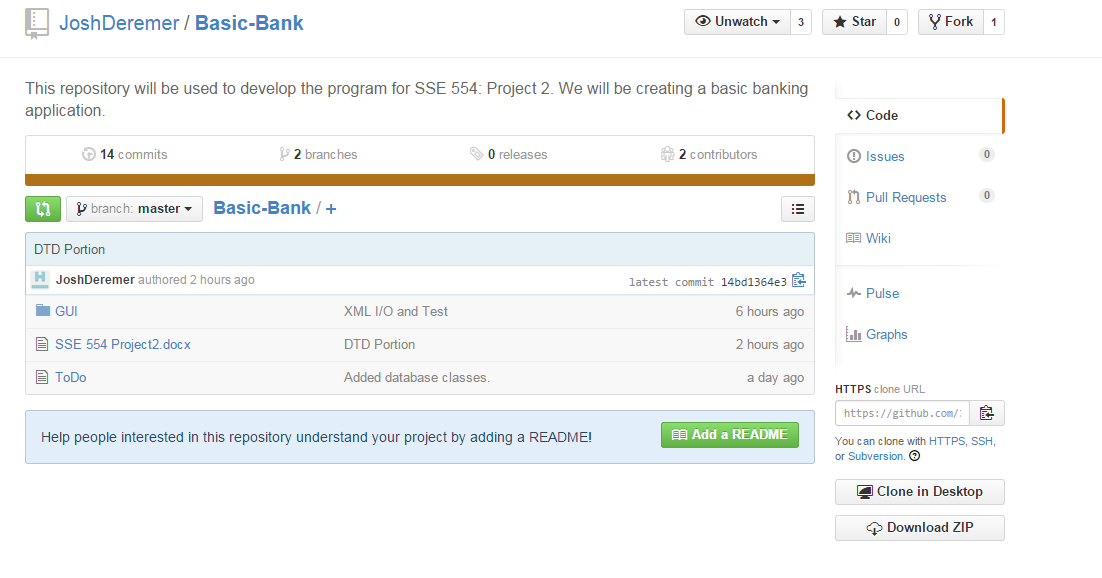
Now lets make a few transactions. Deposit $40.00 and withdraw $50.00.

At this point we have only made changes to the account within the “database” and Array list. In order to write the account details to the XML file. We must back out to the main BankFrame and click on the Exit button. At this point, the writeToFile() method of the Database.java class is called and the following file, if it doesn’t not already exist, is created. It is entitled “Database.xml”. See the figure below.

# Distributed Version Control System

The DVCS was used as central area for all the files related to this project including the report.

## An Picture is worth 1000 words

The group kept track of many things using the repository. Of course the changes in source code were tracked by the dvcs.

The Project report and ToDo list was shared via GitHub in adition to the entire source. It can be found at <https://github.com/JoshDeremer/Basic-Bank>.

# Additional Important Topics

## Using a DTD for XML File Validation

In most circumstances, it becomes essential that XML code has been written in the proper format. In many cases, XML processors are expecting an XML file that has a certain format, including specific tags and attributes associated with those tags. Because of this, different testing methods have been developed for XML files to ensure they are of the proper format. However, as stated earlier, one of the key and powerful advantages of XML is that it is extensible – users can create their own tags to be included in a file. With many different XML files that require a certain set of tags, one XML file may very well require a different set of rules than another. In this situation, a data validation file is practically essential.

There are two different types of validation files that are used often with XML files: Document Type Definitions (DTDs) and XML Schemas. The first, DTD, was utilized in this project to ensure the GUI XML files followed the proper XML standard for tags and attributes that the program was expecting. The difference between these two file types is basic: complexity. DTDs are simple to write, but not very powerful, whereas conversely, XML schema files can be extremely powerful, but become much more complicated.

DTDs contain markup declarations that define document content. A variety of document content rules can be defined by a DTD involving elements, attributes, entities, notations, processing instructions, comments and parameter entity references. The DTD used in this project only focuses on those first two: elements and attributes.

In order to associate a DTD with an XML file, the DTD must be declared directly within the XML file directly after the document declaration using a line such as bellow:  
 <<!DOCTYPE gridbag SYSTEM "gridbag.dtd">  
The gridbag.dtd is the text file that contains the DTD. One thing to note is that a file does not need to be referenced. The rules for the DTD can simply be included within the “!DOCTYPE” tag before closing the tag. DTDs have their own coding format that must be followed so that the XML file and the XML file processor can read the DTD.

Using a DTD with a file can be essential in saving the heartache associated with hunting through the file to find a tag that should not be there or some other form of error. DTDs can help with writing an XML file to check the file along the development path for any discrepancies. In addition, a program that uses the XML file can first compare it to the DTD before attempting to parse the file. If the file is not validated by the DTD, than the program can throw an error or choose not to open the file, both of which are preferable to crashing due to a poorly-formatted XML file.

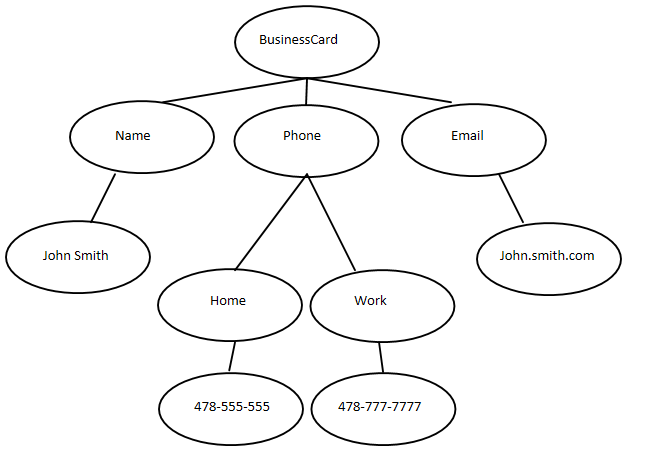
## Using a Document Object Model

Parsing an XML file can be one of the most difficult parts of using XML. You want to be able to read in the information in an orderly manner efficiently. There are multiple methods for doing this, but the one used throughout this project is by creating a Document Object Model (DOM).

A just turns the natural flow of an XML document into a tree structure in memory. Each of the nodes on the tree can be an element or data. The nodes on a tree will have a parent element, child element and sibling elements (although not all nodes have all of these). Consider the following XML code:

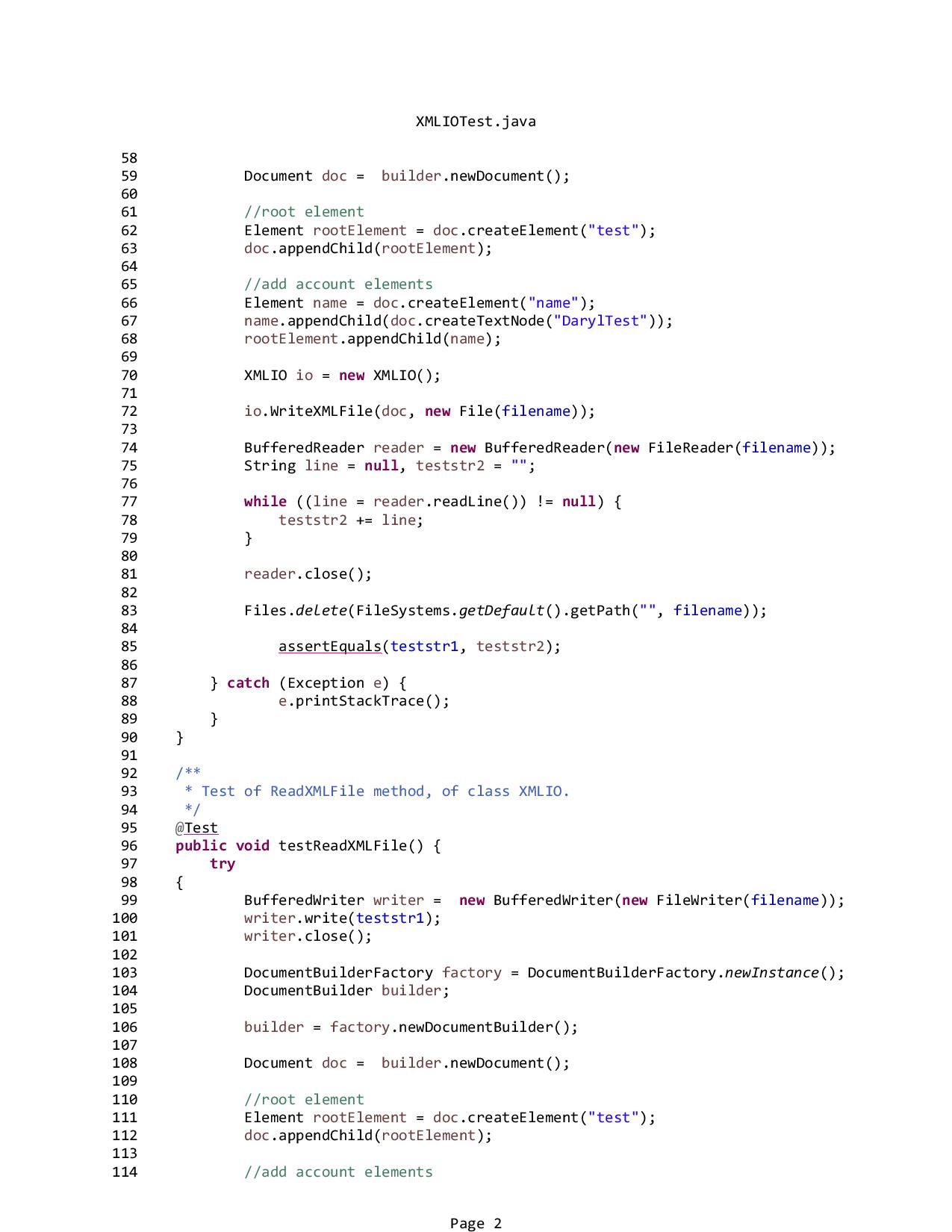
<BusinessCard>  
 <Name>John Smith</Name>  
 <Phone>  
 <Home>478-555-5555</Home>  
 <Work>478-777-7777</Work>  
 <Email>john@smith.com</Email>  
</BusinessCard>

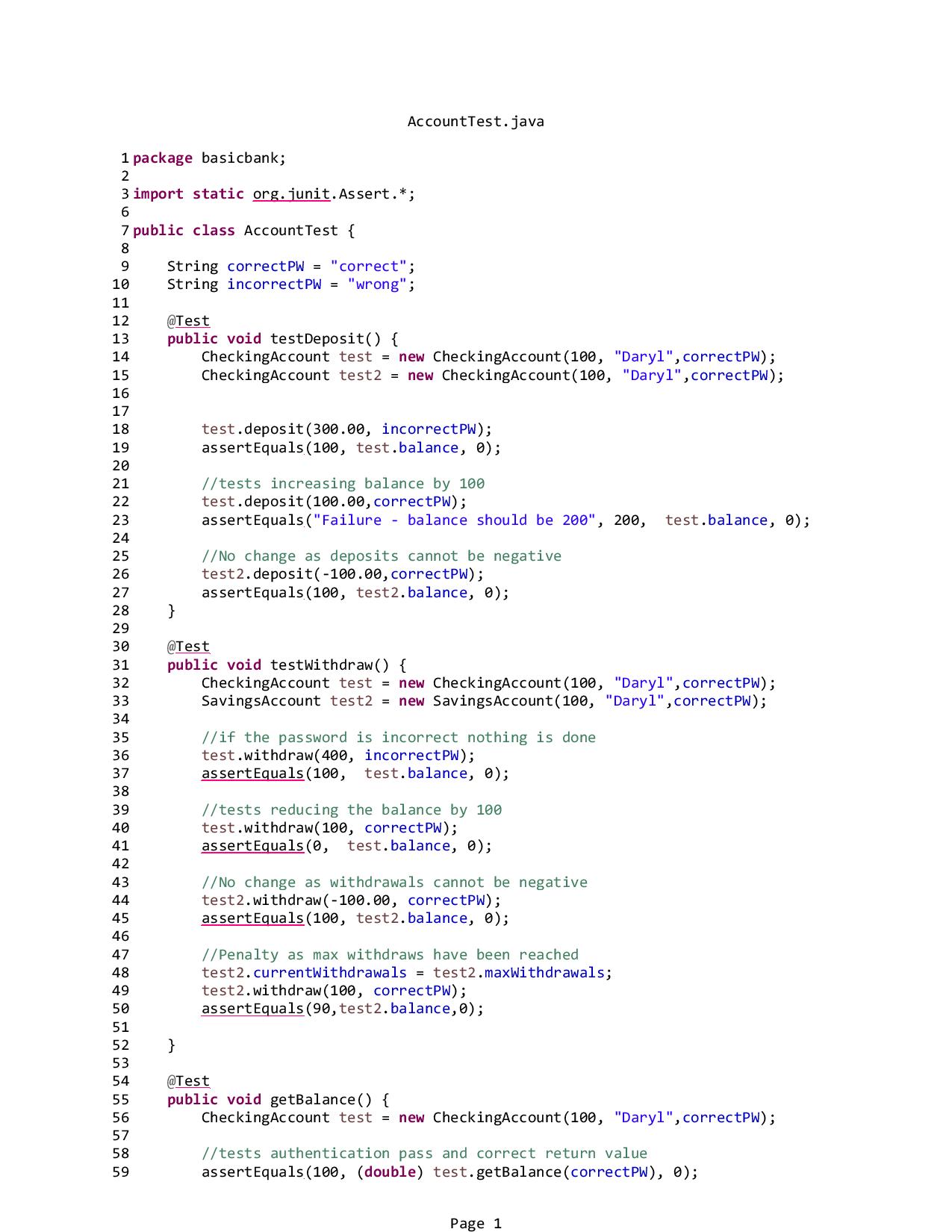
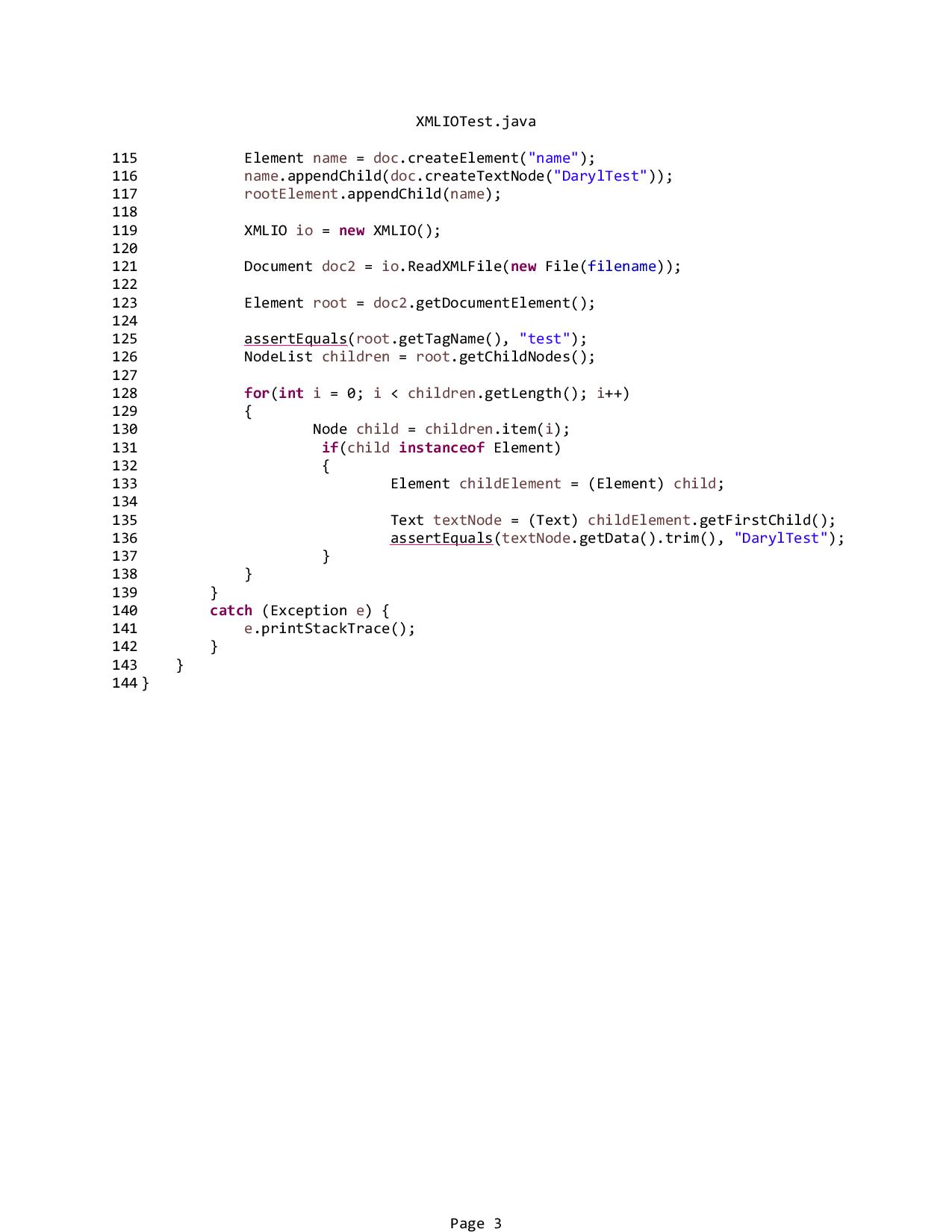
This basic XML code forms a DOM structured as follows:

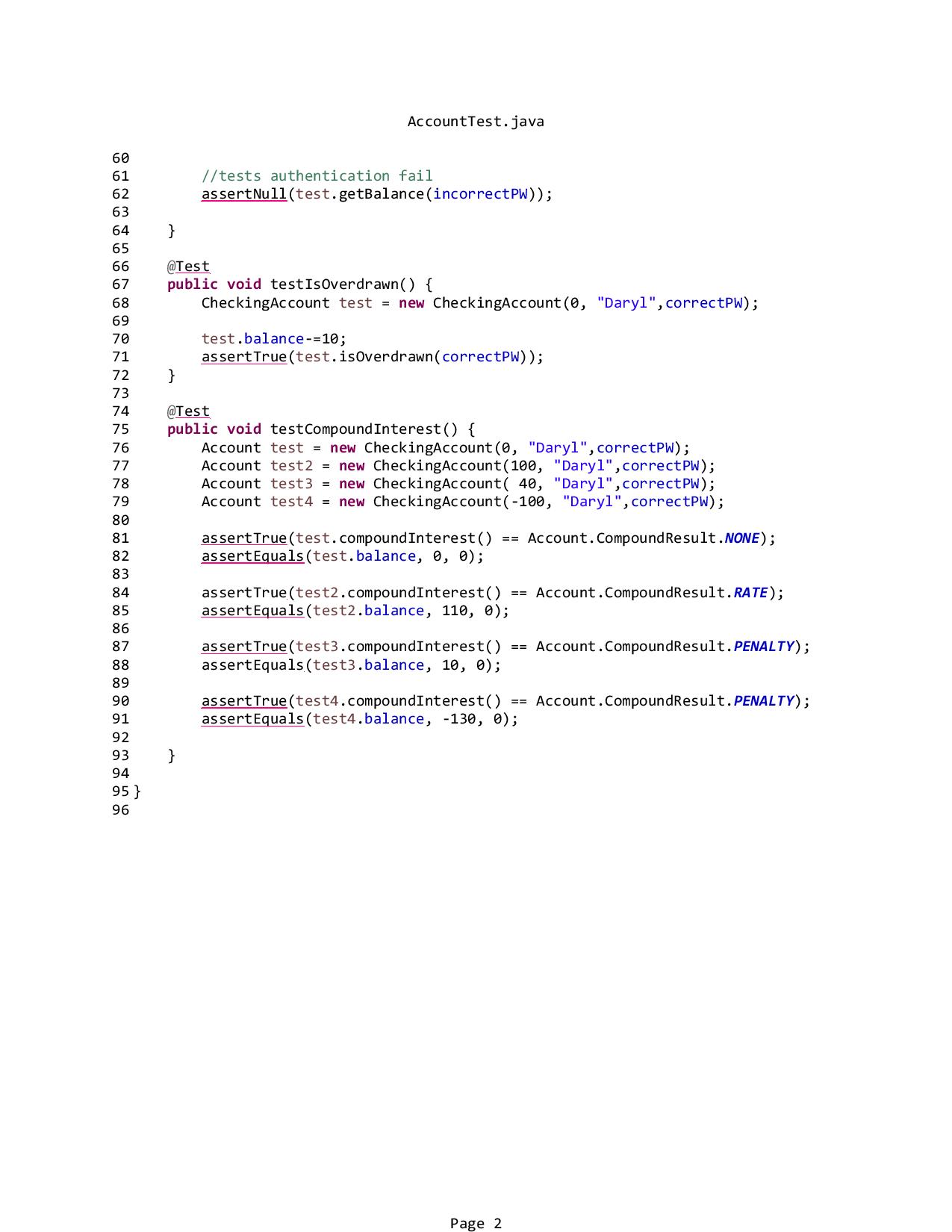


Each oval is a node that can be called. BusinessCard is the root node and has three child nodes: Name, Phone, and Email. This three nodes are siblings of each other and their parent node is BusinessCard. This tree can be traversed by using methods such as getFirstChild(), getNextChild(), getLastChild(), nodeValue(), etc. In addition, the attributes of elements can be retrieved through the getAttribute() method. By employing each of these methods, we are able to traverse through a DOM tree retrieving the information we need from the XML file. DOM trees are generated in this report to pull information in from XML files as well as to write information out to files in the proper XML format.

# E:\Newsbin Downloads\Documents\SSE 554 Project2\XMLIOTest\XMLIOTest-page-001.jpgAppendix I Full Unit Testing Code

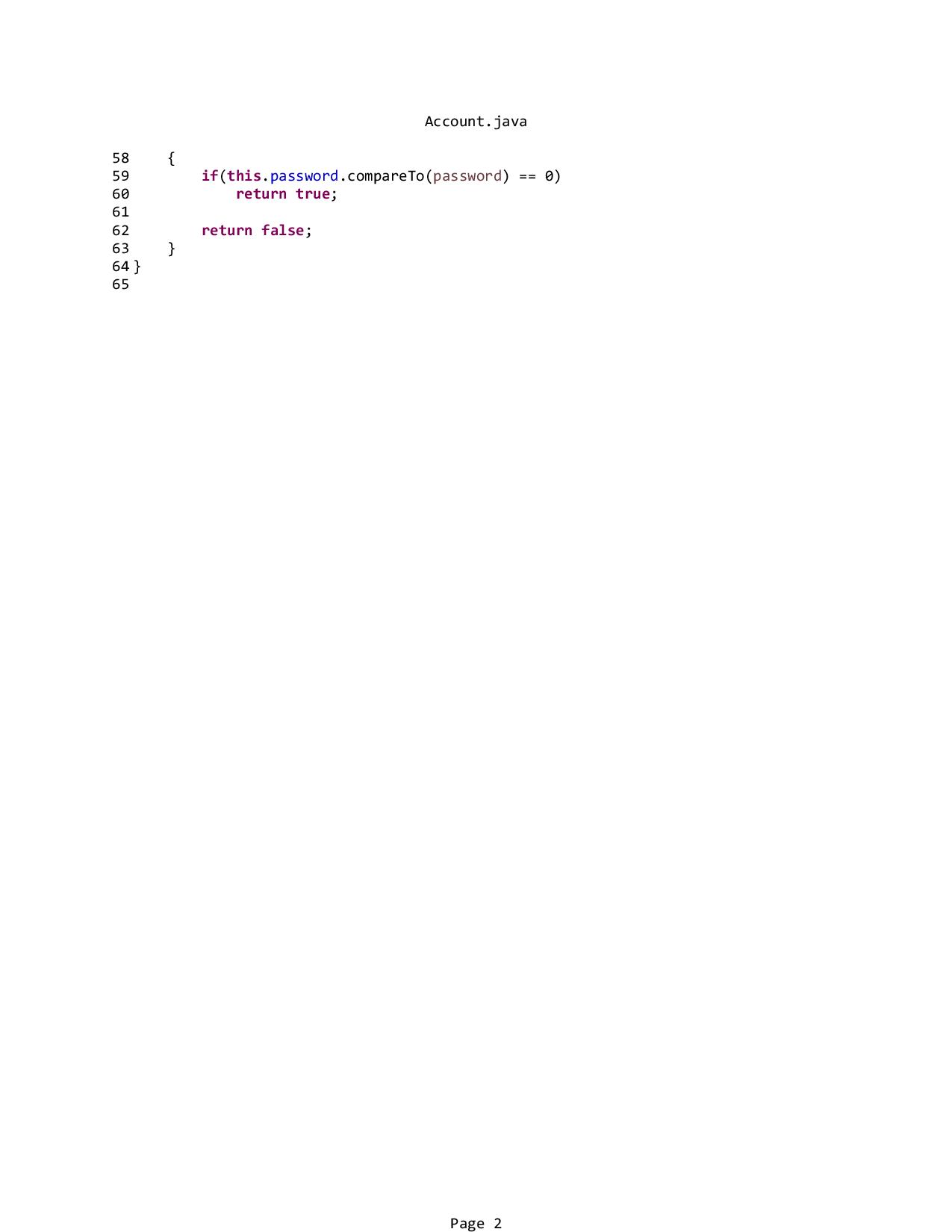


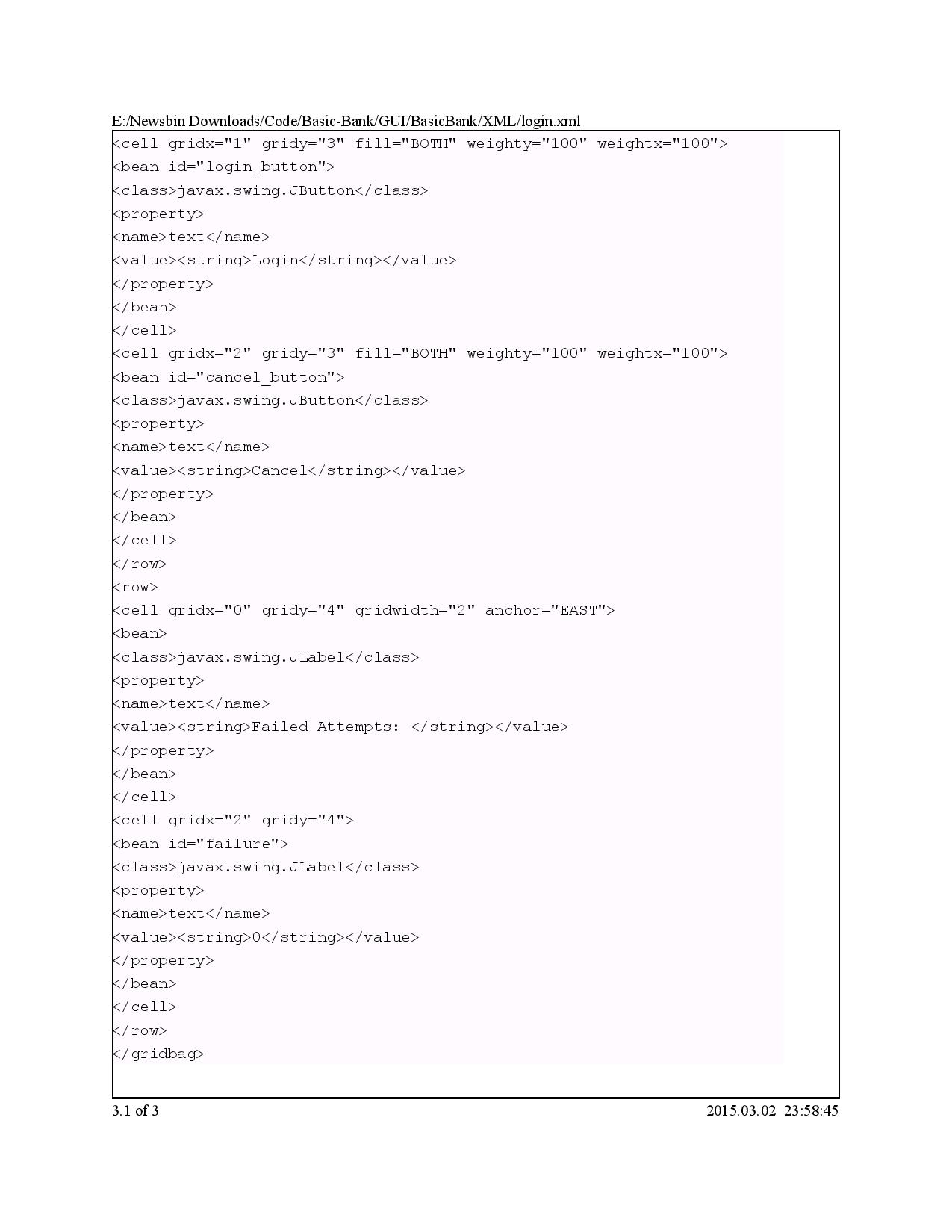


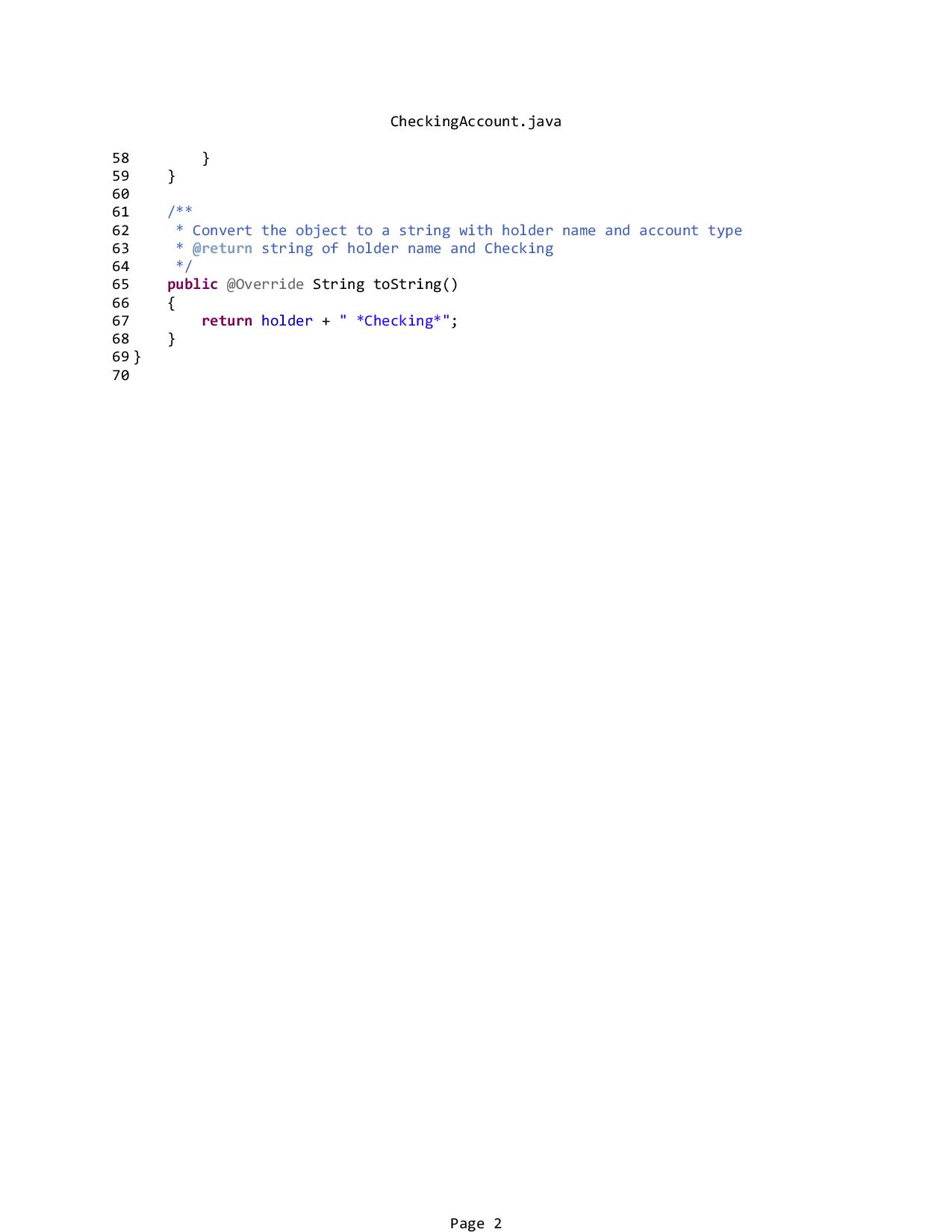


# Appendix II Source Code



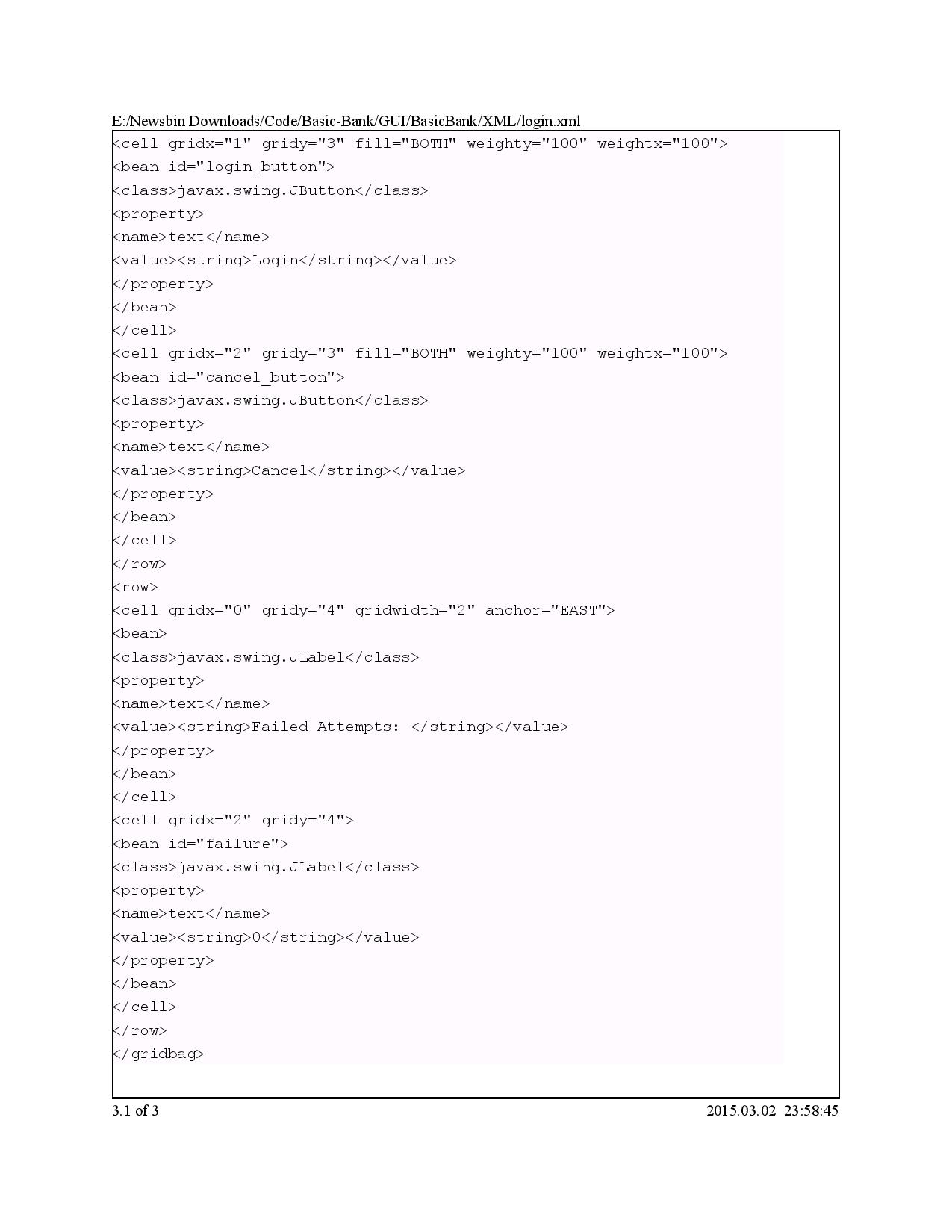


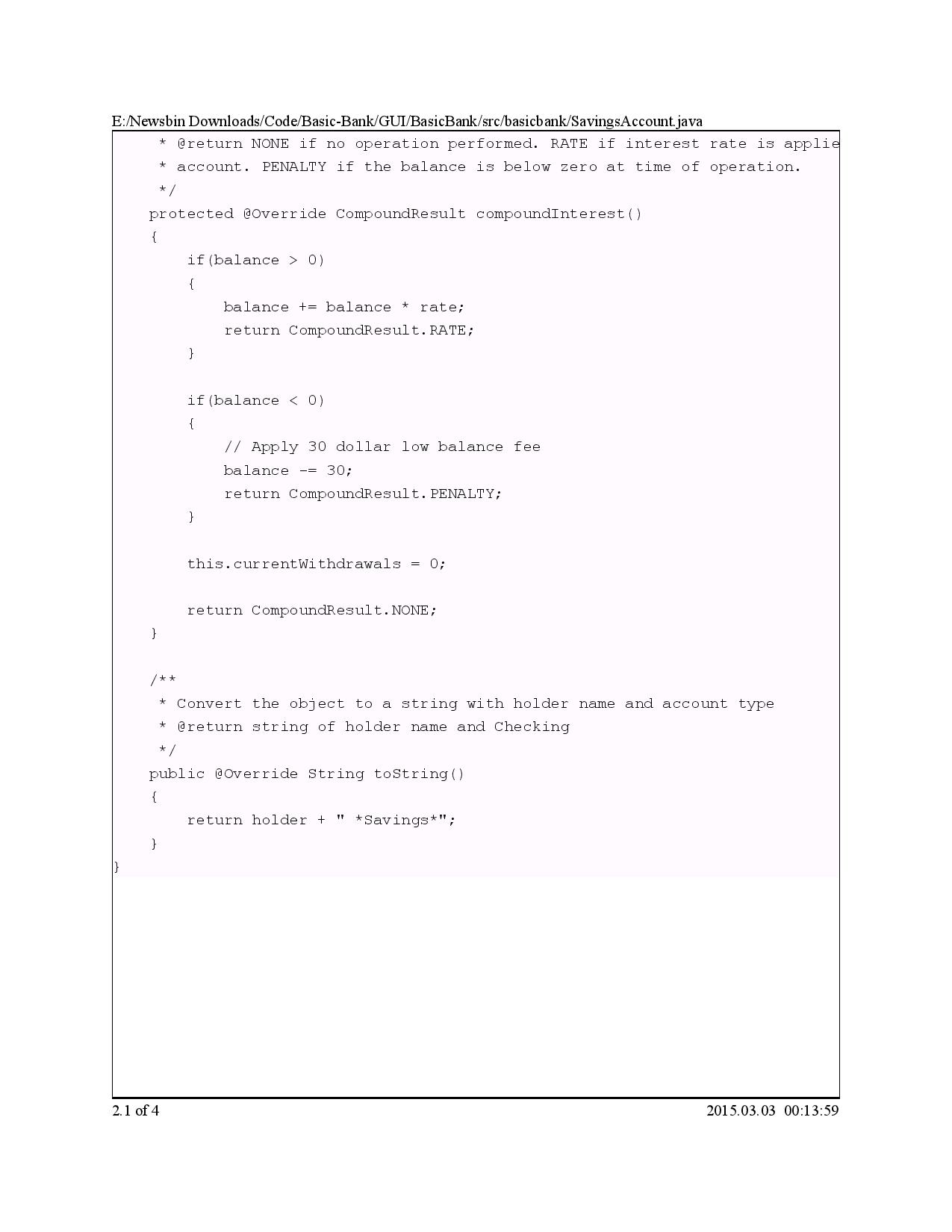
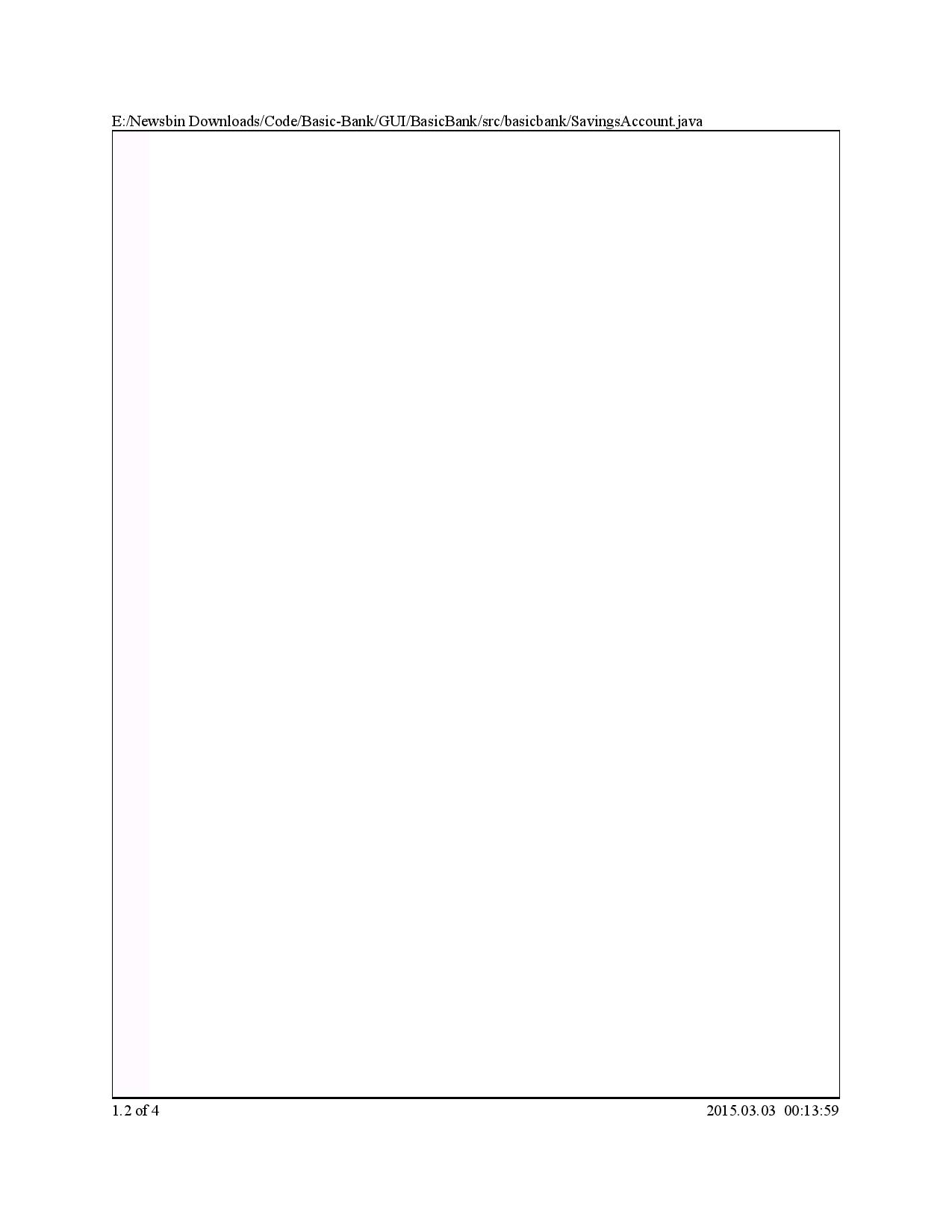
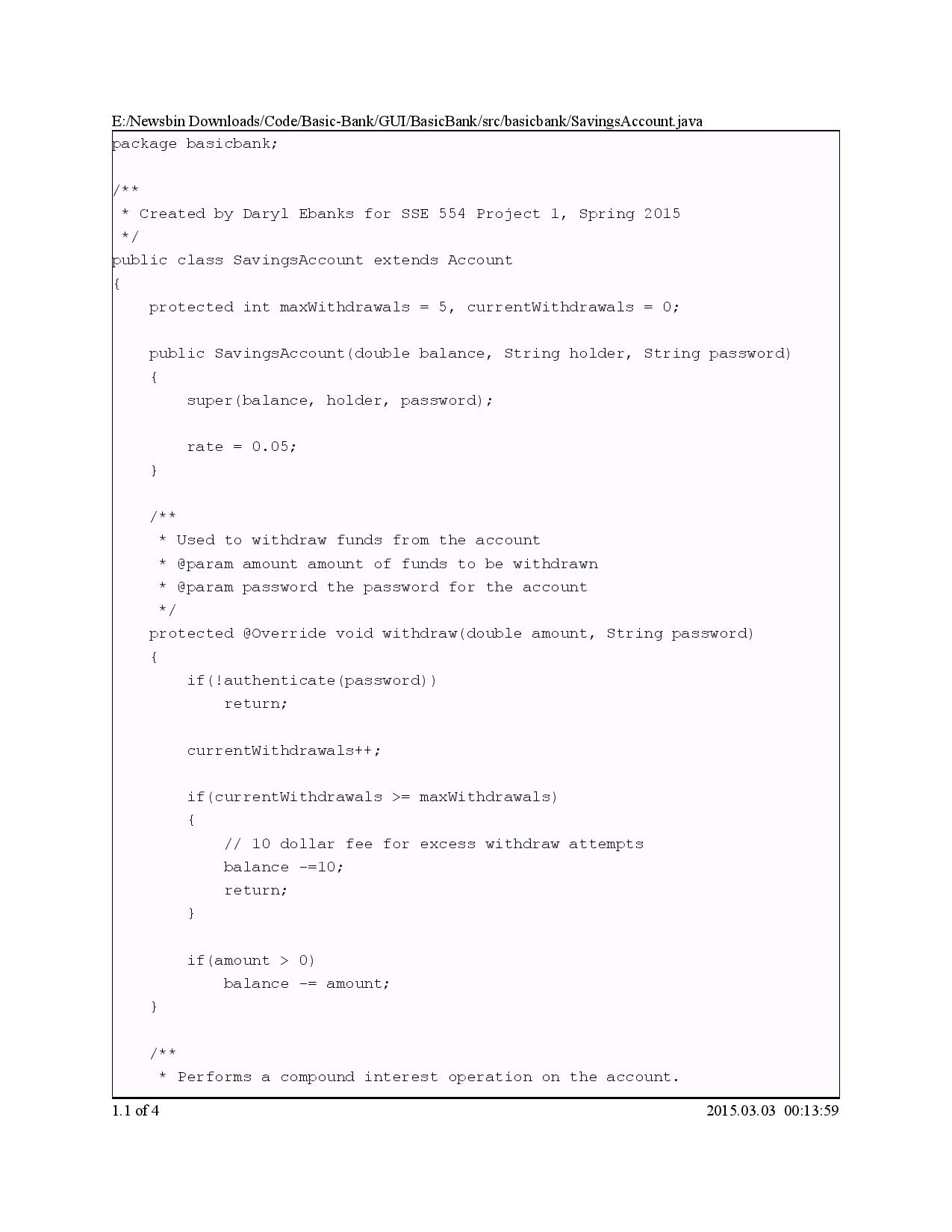


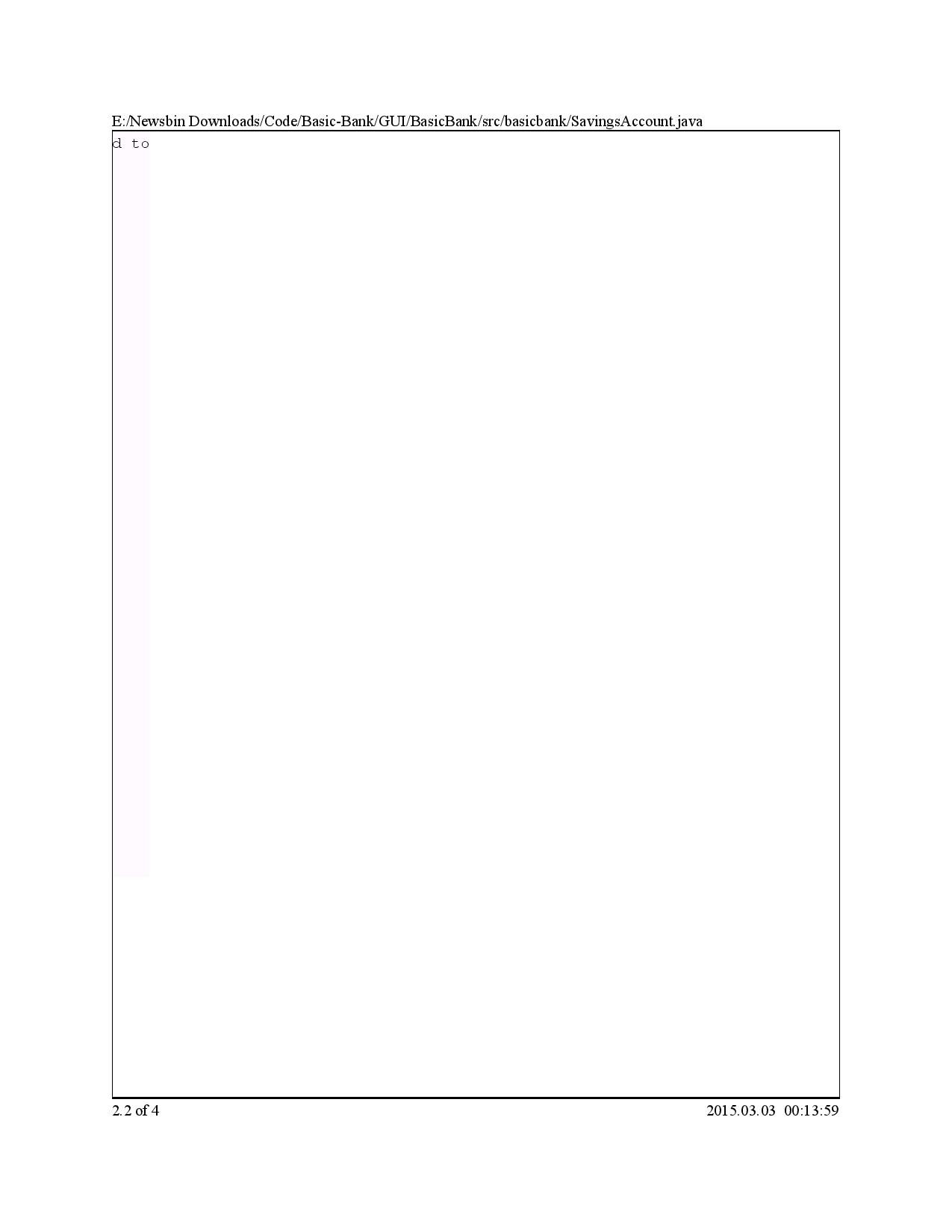












### Database.java

