Java Decisioning Program

Benjamin Roberts

Level 4 Software Development

Contents

[Table of Figures 3](#_Toc169536656)

[Introduction & Requirements 4](#_Toc169536657)

[Project Description 4](#_Toc169536658)

[Acceptance Criteria/Project Outcomes 4](#_Toc169536659)

[Stakeholders Involved 5](#_Toc169536660)

[Development – Set Up Dependencies 6](#_Toc169536661)

[CRUD - Database Connection 6](#_Toc169536662)

[Standing up a Database Solution 6](#_Toc169536663)

[Handing Azure firewall 8](#_Toc169536664)

[Uploading Data 9](#_Toc169536665)

[Access Management 10](#_Toc169536666)

[Personal Data (PII) Protection 13](#_Toc169536667)

[Credentials within the Java program 14](#_Toc169536668)

[Development -Java 16](#_Toc169536669)

[User Input 16](#_Toc169536670)

[Data Model 18](#_Toc169536671)

[Credit Policy built 20](#_Toc169536672)

[Scorecards 20](#_Toc169536673)

[Policy Rules 29](#_Toc169536674)

[Testing the program 30](#_Toc169536675)

[JUnit Unit Tests 30](#_Toc169536676)

[End to End 31](#_Toc169536677)

[Debugging Errors 33](#_Toc169536678)

[Probability of Default - Result always Infinity 33](#_Toc169536679)

[Offers class returning exceptions 35](#_Toc169536680)

[Review with Employer 37](#_Toc169536681)

[Conclusions 38](#_Toc169536682)

[Project Outcome 38](#_Toc169536683)

[Future Additions 38](#_Toc169536684)

[References 39](#_Toc169536685)

[Java Dependencies 40](#_Toc169536686)

[Scorecards Theory & Probability of Default 40](#_Toc169536687)

# Table of Figures

[Figure 1 - A Java Class called “AzureJDBC.java which handles a connection to an Azure SQL Database 8](#_Toc169536622)

[Figure 2 - A screenshot showing a Firewall rule in place on my database to only allow access from my network (with IP Addresses redacted) 9](#_Toc169536623)

[Figure 3 - Teams Messages – Uploading Test data Into Azure SQL Database 10](#_Toc169536624)

[Figure 4 - Azure Data Studio flow for loading data into a database 11](#_Toc169536625)

[Figure 5 - SQL Code showing the creating of the integration user for the Java Program to use 12](#_Toc169536626)

[Figure 6 - SQL Views used within the Java Program 12](#_Toc169536627)

[Figure 7 - SQL Table use to log results history of the Java Program 13](#_Toc169536628)

[Figure 8 - Properties file with the credentials stored in Key-Value pairs, with sensitive information redacted 15](#_Toc169536629)

[Figure 9 - .gitignore file with a record to not include .properties files 15](#_Toc169536630)

[Figure 10 - Implementation of Java reading from an external file to set variables 16](#_Toc169536631)

[Figure 11 - Code used to define the “UserInput” class, importing the necessary Scanner class & defining 17](#_Toc169536632)

[Figure 12 - Java Console showing the Scanner Class in use to capture user input 17](#_Toc169536633)

[Figure 13 - An object method used to prompt the user for a Residential Status 18](#_Toc169536634)

[Figure 14 - the “AppData” class written to act as a Data Model 19](#_Toc169536635)

[Figure 15 - Importing and using the GSON library to create a JSON Representation to log to the console 19](#_Toc169536636)

[Figure 16 - An example of the GSON output, returning a JSON Representation of the Java Program's data model 20](#_Toc169536637)

[Figure 17 - Java Classes for both Scorecards. 21](#_Toc169536638)

[Figure 18 - View of the AppData Class constructor, which the Champion/Challenger split 22](#_Toc169536639)

[Figure 19 - NULL handling for a Scorecard characteristic. 26](#_Toc169536640)

[Figure 20 - Probability of Default formula (Silva et al, 2020). 27](#_Toc169536641)

[Figure 21 - Java Class method to calculate Probability of Default 27](#_Toc169536642)

[Figure 22- SetIndex method for the Champion Scorecard 28](#_Toc169536643)

[Figure 23- SetIndex Method for the Challenger Scorecard 28](#_Toc169536644)

[Figure 24 - Email Chain with colleague for peer review 29](#_Toc169536645)

[Figure 25 - JUnit 5 Test cases written to test individual parts of the program independently 31](#_Toc169536646)

[Figure 26 - A test case written to test a program set up class that reads properties from an external file 32](#_Toc169536647)

[Figure 27 - A demonstration of running the program within the Eclipse Software, with console output 32](#_Toc169536648)

[Figure 28 - Console output for supporting debugging the program 33](#_Toc169536649)

[Figure 29- Unit Test for testing probability of default 34](#_Toc169536650)

[Figure 30 - Unit Test on the Offers class 36](#_Toc169536651)

[Figure 31- Unit Test results showing Illegal Arguement Exceptions 36](#_Toc169536652)

[Figure 32 - Java Switch case missing Break statements 37](#_Toc169536653)

[Figure 33- Correcting the Switch Case, allowing the tests to pass 37](#_Toc169536654)

[Figure 34 - Meeting Minutes for presentation on final version with employer 38](#_Toc169536655)

# Introduction & Requirements

The team I work in is primarily responsible for developing and maintaining a Credit Decisioning software platform. The software we use to do this is provided by a 3rd party and provides a low-code graphic user interface (GUI) for developing business logic into a callable Web service, without the need for specialised IT support. Currently I work as one of the Lead Analysts for this, effectively a Technical Lead for mentoring junior members of the team in building business logic and developing more complex functionality to orchestrate executing business logic e.g. modular designs, integrating database connections & integrating external data sources via HTTP.

This software is built within the Java language, with the GUI interface acting as a layer of separation to allow business users to develop without needing to have experience in the Java language. Personally however, I wish to have a greater understanding of Java, due to a desire to have more experience in object-oriented programming and to better understand how a Java program and various libraries scale to a full application.

With the support of my employer (after a 1-to-1 meeting), I decided to make an attempt to build a similar decisioning application within Java directly. Goals and acceptance criteria have been defined below:

## Project Description

Create a Java program that implements Credit Decisioning logic at an introductory level. This must include:

1. Some ability to capture relevant applicant information required to make a credit decision and enter this information into the program.
2. Execute credit decisioning policy rules that conditionally reject an application if set criteria is met.
3. Connect to an external database to retrieve more data to use for decisioning.
4. Use additional data gathered from this database to execute Scorecards in a champion/challenger approach (requiring at least 2 scorecards to be created)
5. Return a final decision and an offer to the user depending on the decisioning logic.
6. A record of the program run is then saved into an external database (for monitoring and further credit decisioning use)

## Acceptance Criteria/Project Outcomes

1. Final program must have an interface for a user to input application data
2. Application data input must include:
   1. A Match ID of some kind to look up in the external database.
   2. Residential Status
   3. Employment Status
3. A database connection is made to allow CRUD operations for logging activity made in the program.
4. A database connection is made to allow a lookup to be made based on the data received in the program.
5. Multiple credit decisioning policy rules are built and executed in the program flow.
   1. Policy rules are grouped in a modular way to breakdown testing.
   2. A mix of the application input data and data retrieved from the external database is used.
6. Multiple credit decisioning scorecards are built and executed.
   1. A credit offer is made in a champion/challenger style i.e. only 1 is used as the basis for the offer despite both being executed

# Stakeholders Involved

To develop the project, I consulted a number of stakeholders for support. A summary of the input from each stakeholder is recorded below. Communication methods used are also logged as:

* **F** = Face to Face
* **IM** = Instant Messaging via Microsoft Teams
* **C** = Call via Microsoft Teams
* **E** = Email

|  |  |  |
| --- | --- | --- |
| **Stakeholder** | **Methods** | **Input** |
| Credit Risk | **F, IM, C** | * Sign-off for project with Line Manager * Consulted for specific business logic that could be applied to the project & would be appropriate to share * Final review of the project |
| IT Delivery Managers | **F** | * Consulted for advice on how to host the application, specifically for the database link for CRUD operations |
| Asset Management | **E** | * Arranged the installation of all software and licenses required for this project, including:   + Eclipse IDE   + Visual Studio 2022 subscription with Microsoft Azure credits |
| Data Delivery | **IM** | Consulted for:   * advice on setting up an Azure SQL Database user. * importing data into a database that could be used for the program |
| Model Management | **E** | * Fact-checking on the theory sections of the Scorecard Development |

# Development – Set Up Dependencies

## CRUD - Database Connection

In practice, Decisioning Systems like these use a backend database to log application & API call data. They may also store data used in part of the Decisioning Strategy e.g. matching to internal customer data. Creating an integration therefore was a natural requirement. The below sections cover the work required for this.

### Standing up a Database Solution

2 Options were considered for this:

1. MySQL localhost database, a solution I know to be open-source (and therefore free to use)
2. An Azure SQL Database, which was recommended by my employer’s IT Delivery Managers as we hold licences for Visual Studio 2022 subscription, which come with $50 of monthly Azure Credits

The Azure SQL Database was chosen due to:

* The localhost nature of the MySQL solution limited the program to only working on the developer machine.
* Azure representing the opportunity to work with Cloud Computing resources.
* Azure potentially providing a host to deploy this application to if it went further.

Once set up, a connection was established within Java using the MS SQL JDBC library. This is contained within the “AzureJDBC.java” class within my project:

A screenshot of a computer program

Description automatically generated

Figure 1 - A Java Class called “AzureJDBC.java which handles a connection to an Azure SQL Database

### Handing Azure firewall

One issue I had with connecting to the database was that Azure protects its resources with a firewall.

To resolve this, I added my local IP Address to the Firewall rules within Azure to allow devices on my network to interact with the database.

A screenshot of a computer

Description automatically generated

Figure 2 - A screenshot showing a Firewall rule in place on my database to only allow access from my network (with IP Addresses redacted)

### Uploading Data

Credit Decisioning logic typically relies on external data provided by credit bureaus e.g. for the UK those are Experian, TransUnion and Equifax. Being able to retrieve a source for this data was vital to the program.

The scope I set for this project meant that I needed a batch of data the program could access. For this, I used a set of data based on Experian’s DelphiSelect API. This provides users with access to the data Experian hold on a person, which is used to create a credit file for that person. Lenders can then use this to inform their credit decisions (Experian, 2024).

For my program to work, I needed a way to upload a Dataset to the SQL database. For support on this, I consulted our internal Data Delivery team. This was done over Microsoft Teams:

A screenshot of a chat

Description automatically generatedA screenshot of a chat

Description automatically generated

Figure 3 - Teams Messages – Uploading Test data Into Azure SQL Database

From this conversation, 2 solutions for uploading a dataset based off this API into the Database were identified:

1. **Import Wizard** via the Azure Data Studio Software, the main software used for working with Azure Databases
2. **Azure Data Factory** – another cloud-based tool offered by Microsoft for ETL (Extract, Transform, Load) Data flows.

The Import Wizard was preferred due to its simplicity, but this kept timing out while loading. Azure Data Factory was the alternative I discussed with our Data Delivery team (see Appendix). The tool provides a beginner friendly “Ingest” template, which I was able to use to get my test dataset loaded.

Shown below is an example of this in action. The pipeline was run in “Debug” mode, which caters for one-off scenarios like this,

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

Figure 4 - Azure Data Studio flow for loading data into a database

### Access Management

When creating the database, I was required to create a root/admin user, with full permissions to edit the database. It’s not appropriate to use this for my application as it has far more permissions than it actually needs, introducing security risks (Snowflake, 2024).

To set an appropriate level of access, I created a new Integration User within my database & granted only permissions that were needed.

A screenshot of a computer program

Description automatically generated

Figure 5 - SQL Code showing the creating of the integration user for the Java Program to use.

Permissions granted to this user included:

* SELECT access to the views required by the program to execute (Microsoft, 2024)

A screenshot of a computer

Description automatically generated

A black background with white text

Description automatically generated

Figure 6 - SQL Views used within the Java Program

* INSERT access to the “dbo.JavaDecisioningHistory” table where transaction history will be logged.

A screenshot of a computer program

Description automatically generated

Figure 7 - SQL Table use to log results history of the Java Program

### Personal Data (PII) Protection

This access management configuration can also be used for another purpose: protecting Personal Data (PII). Credit Decisioning Systems like this must collect personal data to be able to conduct a credit search on a person e.g. Experian’s DelphiSelect API requires at least:

* Full name of the person in question
* Date of Birth
* At least the current address of the person in question
* Previous addresses are also often needed if the person in question has resided at their current address for less than 3 years

(Experian, 2024)

Each of these Data Points are protected under the UK's Data Protection Act 2018, which also codifies the GDPR into UK Law (Data Protection Act, 2018). The program is currently collecting the Full Name of the user as a starting point to allow a full API integration in future. This means the program must be mindful of Data Protection/GDPR requirements.

Using a SQL databases within the program creates the risk of SQL Injection, where malicious input into the program can lead to arbitrary execution e.g. a user passes in a SQL command instead of a name (W3Schools, 2024). This is where the SELECT and INSERT permissions assigned above come into effect. The Full Name collected in the application is only saved to the “dbo.JavaDecisioningHistory” , which the integration user only has INSERT permissions on. This prevents a malicious user from entering a query to retrieve data from this table, as the database permissions will result in the query being rejected.

The only other permission the Integration User has been granted is the SELECT permission on the “Delphi.VW\_DelphiPremiumValueData” and the “Delphi.VW\_DelphiSummaryData” views. These are the only other views required by the program, so by limiting the access like this the possibility of risk incurred by SQL Injection is minimised and therefore acts as a control to protect Personal Data.

### Credentials within the Java program

To avoid hard coding these credentials within the program, a “.properties” file was created:

A screenshot of a computer

Description automatically generated

A screen shot of a computer

Description automatically generated

Figure 8 - Properties file with the credentials stored in Key-Value pairs, with sensitive information redacted

For security purposes, this properties file is not committed to the GitHub repository used for this project. It falls into a general rule to exclude all .properties files from the repository. This is done by adding a record to the “.gitignore” file, which tells Github files matching the names or expressions within should not be monitored. The \*.properties line acts as a catch-all to filter out all properties files like this.

A screen shot of a computer

Description automatically generated

Figure 9 - .gitignore file with a record to not include .properties files

Java can read these files by the “key” names e.g. for this file, Java can reference “jdbcUser” within the program and extract the value. The advantages of this are:

* Credentials are not hard coded, making them easy to change
* They can be stored in files that are not tracked by Version Control software like Github.

Below is code showing how this is implemented in the program:

A screen shot of a computer program

Description automatically generated

Figure 10 - Implementation of Java reading from an external file to set variables

The try-catch logic is used to catch scenarios where the config properties filed cannot be found. Without this, the program can’t run. To avoid any unnecessary behaviour if this occurs, the “System.Exit” method is used to terminate the program immediately.

# Development -Java

## User Input

The Java Scanner class was used to prompt the user for input. Defined below is a class called “UserInput” which has been written to group all methods related to capturing user input together:

A screenshot of a computer program

Description automatically generated

Figure 11 - Code used to define the “UserInput” class, importing the necessary Scanner class & defining

Upon running the program, the user will be asked a series of questions:

1. Their Forename
2. Surname
3. The ResidentialStatus and their EmploymentStatus, presented as a menu.

Console output for this is below:

A screenshot of a computer

Description automatically generated

Figure 12 - Java Console showing the Scanner Class in use to capture user input

An example of the code that generates this console output is annotated below:

A screen shot of a computer program

Description automatically generated

Figure 13 - An object method used to prompt the user for a Residential Status

What each part of this does:

* “println()” functions print out a menu of options to the console for the user
* A custom defined “getIntInput” method calls the Java Scanner & writes out another prompt to the console with instructions to the user
* A “while” loop traps the user in this loop until they provide a valid option
* The “try-catch” block is included to capture invalid characters e.g. string values when expecting a value of 1 to 6
* The “loopCount” increment logic is included as a failsafe for ending the while loop in case of too many bad inputs and/or preventing an infinite loop

## Data Model

One issue I ran into early on was how to logically structure the data used within the program. When working with Credit Decisioning systems in the workplace, we use a Data Model to define what data the application has access to and how it is stored.

I chose to apply a similar approach to this program, by defining an “AppData” class:

A screenshot of a computer

Description automatically generated

Figure 14 - the “AppData” class written to act as a Data Model

Doing this gave these advantages:

1. All application data is logically grouped
2. A single data object can be logged for review
3. Java Methods could be written to accept this model, allowing some flexibility around Java’s strongly typed data constraints when writing methods

Once the program completes its run, this Data model is populated with all the data used during the run. This allows the data to be logged easily for review. To do this, I made use of the GSON library to get a JSON representation of this:

**A screen shot of a computer

Description automatically generated**

Figure 15 - Importing and using the GSON library to create a JSON Representation to log to the console

An abridged example output of this is below:

{

"appl": {

"forename": "T",

"surname": "T",

"employmentStatus": "FullTime",

"residentialStatus": "HomeOwner"

},

"decData": {

"pk": 4874,

"E1B07": "0",

"E1B08": "0",

"E1B09": 2,

"TRD\_A\_13": -3,

"E1A09": 0,

"E5S051": 777,

"NDSPCII": -4,

"TRD\_A\_06": 2,

"E1B13": 0,

"E1B01": 0,

"TRD\_STL\_14": -2,

"TRD\_STL\_19": -2

},

"scorecardRngResult": 77,

"strategyPath": "CHALLANGER",

"scorecards": [

{

"scorecardName": "ChampionScorecard",

"score": 733,

"index": 0,

"probOfDefault": 0.0024726231566347745,

"stringCharacteristics": [

{

"characteristicValue": "0",

"name": "E1B07",

"bandingScore": 27

},

{

"characteristicValue": "HomeOwner",

"name": "ResidentialStatus",

"bandingScore": 40

}

],

"numberCharacteristics": [

{

"characteristicValue": 2,

"name": "E1B09",

"bandingScore": 10

},

{

"characteristicValue": -3,

"name": "TRD-A-13",

"bandingScore": 40

},

{

"characteristicValue": 0,

"name": "E1A09",

"bandingScore": 46

},

{

"characteristicValue": -2,

"name": "TRDSTL14",

"bandingScore": -30

}

]

},

{

"scorecardName": "ChallangerScorecard",

"score": 691,

"index": 0,

"probOfDefault": 0.01798620996209156,

"stringCharacteristics": [

{

"characteristicValue": "FullTime",

"name": "EmploymentStatus",

"bandingScore": 45

}

],

"numberCharacteristics": [

{

"characteristicValue": -4,

"name": "NDSPCII",

"bandingScore": -40

},

{

"characteristicValue": 2,

"name": "TRD-A-06",

"bandingScore": 14

},

{

"characteristicValue": 0,

"name": "E1B13",

"bandingScore": 26

},

{

"characteristicValue": 0,

"name": "E1B01",

"bandingScore": 34

},

{

"characteristicValue": -2,

"name": "TRD-STL-19",

"bandingScore": 32

}

]

}

],

"flags": [

{

"policyCode": "D\_001",

"ruleOutcome": "PASS",

"decisionType": "DECLINE"

},

{

"policyCode": "D\_002",

"ruleOutcome": "PASS",

"decisionType": "DECLINE"

},

{

"policyCode": "D\_003",

"ruleOutcome": "PASS",

"decisionType": "DECLINE"

},

{

"policyCode": "A\_001",

"ruleOutcome": "PASS",

"decisionType": "ACCEPT"

}

]

}

Figure 16 - An example of the GSON output, returning a JSON Representation of the Java Program's data model

## Credit Policy built

The below section details the Credit Decisioning Logic built into the program.

### Scorecards

Credit Scorecards are a tool in credit risk management used by Credit Risk Management teams to model/predict the likelihood of default for a given subject borrower (i.e. an applicant/customer) over a period (Huang and Scott, 2017). They are used for evaluating the creditworthiness of a customer, making them critical part of the Credit Decisioning process. This made them a natural choice to build in this project.

Scorecards from a program point of view are made up of:

* Characteristics which receive a score based on their value
* A final calculated score based on the sum of the characteristic scores + a starting score
* A “Score Index” used to group ranges of scores together. Typically translates into groups the business can use to simply communicate which applicants are more risky
* A “Probability of Default”, which translates the raw score into a percentage chance the applicant would default on a credit offer i.e. they are unable to pay

This project implements 2 scorecards in a champion/challenger fashion. This is an approach that deploys multiple competing strategies into a production environment and monitoring which provides the best performance (FICO, 2020). Using this is an effective demonstration of using Object-Oriented programming to build Credit Decisioning logic.

Note that both scorecards are not based on any actual statistical model. They are only for demonstrating how the concept can be implemented in an Object-Oriented program.

The Characteristics form the core of any scorecard. Given that each scorecard is unique in what variables it uses and how it assesses each variable, I chose to use classes containing static methods to implement these. This is shown in the figure below:

A screen shot of a computer

Description automatically generated

Figure 17 - Java Classes for both Scorecards.

#### Decide between Champion and Challenger

For a Champion/Challenger strategy, the goal is to send a percentage of applicants down the challenger path and compare the outcomes to the champion. To start, most of the volume would be directed to the known good model (i.e. the champion). This allows the business to test the challenger model with a limited volume to analyse if better business outcomes are given by the challenger.

To apply this to my program, I added logic into my AppData model constructor to decide which strategy path the program should follow. This logic is below:

A screen shot of a computer program

Description automatically generated

Figure 18 - View of the AppData Class constructor, which the Champion/Challenger split

Defining this logic in the constructor of the Data Model allows me to easily ensure this split is made early in the program. It is implemented as:

* 60% of the users will use the Champion Path
* 40% use the Challenger Path

The “Random” Java class is used to generate a random number to enforce this split.

#### Characteristics

These scorecards consist of a starting/base score plus a series of characteristics, variables with values that represent a statistical insight into the applicant/customer e.g. the E1A09 characteristic in the Champion scorecard count how many delinquent (i.e. in arrears) account they have present on their account (Experian, 2024). Specific values in that variable get assigned Banding Scores (or Points) which represent if they are a positive or negative indication of the applicant’s credit worthiness.

Once calculated, all the Banding scores are added together, along with an initial starting/base score to calculate a final score for the applicant, which represents their credit worthiness. A financial institution can use this as a basis for deciding a credit offer i.e. do they offer them a product and what terms they will offer if they do.

Below are specifications of both the Champion and Challenger scorecards built into the program, documented in a style typical in the workplace. The columns mean:

* **Characteristic** = the name of the variable used. Can reference ether data the applicant enters on an application form or data available on their Credit records accessed via Credit Bureaus like Experian
* **Data Type –** documents the type of values the characteristic can hold. Useful for guiding developers in how to handle the values within
* **Min Value & Max Value –** These relate to the **Points** column & defines the ranges used to set a specific Banding Score/Points. For String values, this is a 1-to-1 mapping for value to points
* **Points –** The actual Banding score assigned to the characteristic, which is then added to the overall score on the scorecard

#### Champion Specification

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Characteristic** | **Data Type** | **Min Value** | **Max Value** | **Points** | **Notes** |
| Base Score | Numeric | [default] | | 600 | This is a static value that all instances of this scorecard will start from. |
| E1B07 | String | T | | 0 | External data provided by Experian’s Credit Record API: DelphiSelect  Defined as: ***Worst Status in the last 6 months of all Active CAIS accounts (SP)***   (Experian, 2024). |
| N | | 0 |
| D | | 0 |
| U | | 0 |
| 0 | | 27 |
| 1 | | 14 |
| 2 | | -10 |
| 3 | | -23 |
| 4 | | -23 |
| 5 | | -23 |
| 6 | | -23 |
| 8 | | -50 |
| [default] | | -50 |
| E1B09 | Numeric | -1 | 0 | -20 | External data provided by Experian’s Credit Record API: DelphiSelect  Defined as***: Number of Active CAIS Accounts (SP)*** (Experian, 2024). |
| 1 | 2 | 10 |
| 3 | 4 | 20 |
| 5 | 6 | 30 |
| 7 | max | 18 |
| [default] | | -20 |
| TRD-A-13 | Numeric | -3 | -3 | 40 | External data provided by Experian’s Credit Record API: DelphiSelect  Defined as***: Number of Months Since Most Recent Arrears on all Accounts allowed*** (Experian, 2024). |
| -2 | -2 | -20 |
| -1 | 0 | -20 |
| 1 | 1 | 10 |
| 2 | 2 | 5 |
| 3 | max | -30 |
| [default] | | -30 |
| E1A09 | Numeric | -1 | -1 | 0 | External data provided by Experian’s Credit Record API: DelphiSelect  Defined as: ***Number of Delinquent CAIS Accounts, excluding Mail Order accounts*** (Experian, 2024). |
| 0 | 0 | 46 |
| 1 | 2 | 24 |
| 3 | max | -26 |
| [default] | | -26 |
| TRD-STL-14 | Numeric | -2 | -2 | 26 | External data provided by Experian’s Credit Record API: DelphiSelect  Defined as: ***TSMR Active Short Term Loans account opened (Active CAIS)*** (Experian, 2024). |
| -1 | -1 | 0 |
| 0 | 0 | 26 |
| 1 | 6 | -24 |
| 7 | 18 | -15 |
| 19 | 36 | 6 |
| 36 | max | 15 |
| [default] | | -24 |
| ResidentialStatus | String | HomeOwner | | 40 | This data point is one of the User Input questions.   The status indicates the type of accommodation someone lives in. |
| PrivateTenantFurnished | | 26 |
| PrivateTenantUnfurnished | | 34 |
| CouncilTenant | | 10 |
| Cohabiting | | 34 |
| LivingWithParents | | 10 |
| [default] | | 10 |

#### Challenger Specification

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Characteristic** | **Data Type** | **MIn Value** | **Max Value** | **Points** | **Notes** |
| Base | Numeric | [default] | | 580 | This is a static value that all instances of this scorecard will start from. |
| NDSPCII | Numeric | -3 | -1 | 0 | External data provided by Experian’s Credit Record API: DelphiSelect  Defined as: ***Consumer Indebtedness Index (SP)*** (Experian, 2024). This is an index that represents how in debt a person is. Higher values indicate the subject is using more of their available credit. |
| 1 | 10 | 45 |
| 11 | 20 | 23 |
| 21 | 25 | 14 |
| 26 | 40 | 5 |
| 41 | 50 | -20 |
| 51 | max | -40 |
| default | | -40 |
| TRD-A-06 | Numeric | -2 | -2 | 0 | External data provided by Experian’s Credit Record API: DelphiSelect  Defined as: ***Number of Consistently up-to-date accounts L12m*** (Experian, 2024). |
| -1 | -1 | -10 |
| 0 | 0 | -20 |
| 1 | 1 | 6 |
| 2 | 2 | 14 |
| 3 | max | 21 |
| default | | -20 |
| E1B13 | Numeric | -1 | -1 | -10 | External data provided by Experian’s Credit Record API: DelphiSelect  Defined as: ***Number of CAIS Status 3 or worse, within the last 6 Months for all Active CAIS accounts (SP)*** (Experian, 2024). CAIS Status refers to the arrears position of the account e.g. 3 means 3 missed payments on the account. |
| 0 | 0 | 26 |
| 1 | 1 | -12 |
| 2 | max | -56 |
| default | | -56 |
| E1B01 | Numeric | -1 | -1 | 0 | External data provided by Experian’s Credit Record API: DelphiSelect  Defined as: ***Number of Active CAIS Accounts Opened in the Last 3 Months (SP)*** (Experian, 2024). |
| 0 | 0 | 34 |
| 1 | 1 | 12 |
| 2 | 2 | -11 |
| 3 | max | -33 |
| default | | -33 |
| TRD-STL-19 | Numeric | -2 | -2 | 32 | External data provided by Experian’s Credit Record API: DelphiSelect  Defined as: ***Total Number of Default Short Term Loans accounts (Default CAIS)*** (Experian, 2024). |
| -1 | -1 | 0 |
| 0 | 0 | 28 |
| 1 | 1 | -14 |
| 2 | max | -27 |
| default | | -27 |
| EmploymentStatus | String | FullTime | | 45 | This data point is one of the User Input questions.   The status indicates the type of employment an applicant is in. |
| PartTime | | 24 |
| Student | | 6 |
| SelfEmployed | | 33 |
| Unemployed | | -10 |
| Retired | | 22 |
| default | | -10 |

#### NULL Handling in characteristics

NULL values are technically possible within the External Data variables detailed in the specifications above. In this program the expectation is that those should fall into the “default” bands and get assigned the relevant points.

To handle this in Java, try-catch blocks are used to specifically catch a NullPointerException that would arise from attempting to use these values e.g. the screenshot below shows this applies to the E1B09 characteristic on the Champion scorecard:

A screen shot of a computer program

Description automatically generated

Figure 19 - NULL handling for a Scorecard characteristic.

#### Probability of Default formula

The calculated score on a Scorecard is intended to be a representation of how likely a subject is likely to default on any offered credit. In its natural form however, this isn’t a clear value to communicate probability to non-technical stakeholders. This becomes important when collaborating with product teams in credit decisioning, who typically do not have the subject matter expertise to interpret raw credit scores. The probability of default works as a more effective communication tool to help them set product offers e.g. max loan amounts and terms.

The Probability of Default formulas allow us to translate this raw score into a percentage value that can be more cleanly communicated to non-technical stakeholders. It makes use of coefficients (called “beta” β) and model variables defined early on in the Scorecard Model Development process to convert the raw score into a percentage value.

An accepted model for calculating this value (based on Logical Regression) is provided below:

|  |  |
| --- | --- |
| Probability of Default formula | **e** = Eular’s Number (i.e. exponent)  **β** = beta (i.e. the model coefficients)  **x** = Model variables defined in the Scorecard Model Development process |

Figure 20 - Probability of Default formula (Silva et al, 2020).

Beta i.e**. β** is a static value defined as part of the development process. Doing so requires using historical data on the population we’re modelling for. This data is run through statistical analysis (e.g. logical regression) to estimate a co-efficient to use.

X represents model variables defined in the scorecard process. For the purpose of this mock model, those are our Scorecard Characteristics covered above. The Points assigned to each variable are added together to form this.

Defining the constants to use in this model is typically done in the model development process. As only mock scorecards are used for this program to demonstrate the scorecards concept, the output of this will have no statistically meaningful value. The focus is solely on the Java implementation.

This screenshot below shows this coded in the “Scorecard.java” class:

A screen shot of a computer program

Description automatically generated

Figure 21 - Java Class method to calculate Probability of Default

Where:

* Math.exp() function provides the Exponent value required for the formula
* baselineScore & pointsJump represent the constants to be uses in the formula

#### Score Indexes

Score indexes are another method of translating the raw score from the Scorecard into an easier to understand metric for non-technical stakeholders. They help group ranges of scores/probability of defaults together to allow other business teams to make decisions on how they treat the person in question. One key example of this is deciding terms for offering a person a credit agreement e.g. how much money the business is prepared to lend over a set period of time (months, years etc).

To demonstrate this, I’ve coded “setIndex” methods into both my Champion and Challenger scorecards:

A computer screen shot of a program

Description automatically generated

Figure 22- SetIndex method for the Champion Scorecard

A computer screen shot of a program

Description automatically generated

Figure 23- SetIndex Method for the Challenger Scorecard

#### Peer Review

Given the complexity of the Scorecard Development process, I wanted to have the information above peer reviewed to ensure accuracy. For this I reached out to our internal Model Monitoring team. They act as an Internal Audit team who reviewing the performance and development process of how models like these Scorecards are built, making them perfect technical stakeholder contact for validating this information.

Below is the email chain that came from this:

Hi Ben,

I have modified some wordings in the scorecard section and added my comments on the probability of default formula(p12).

Hope this helps.

Kind Regards,

[Colleague]

**From:** Benjamin Roberts (Credit) <Benjamin.Roberts@vanquis.com>  
**Sent:** Wednesday, May 29, 2024 1:53 PM  
**To:** [Colleague]

**Subject:** Re: Scorecard training docs?

Thanks [Colleague]🙂

Could I take you up on that please?

One thing I need to do for the project is explain the concept of a scorecard & the parts of it. I've put something together in the attached word doc from my basic understanding.

Could you give the Scorecard section a read please & fact check it?

**Sent:** Monday, May 20, 2024 15:07  
**To:** Benjamin Roberts (Credit) <[Benjamin.Roberts@vanquis.com](mailto:Benjamin.Roberts@vanquis.com)>  
**Subject:** RE: Scorecard training docs?

Hi Ben,

It’s a while back (maybe 9 years ago) when [Colleague] and people at that year joined the company I did run some model/scorecard development training courses. I don’t know if I keep those and have moved them to SharePoint as PFG had been transformed greatly. I will do a digging to see if I could find anything.

In the meantime, I am very happy to help you to understand any steps/procedures on the Scorecard development life cycle if you have any questions.

Kind Regards,

[Colleague]

**From:** Benjamin Roberts (Credit) <[Benjamin.Roberts@vanquis.com](mailto:Benjamin.Roberts@vanquis.com)>  
**Sent:** Monday, May 20, 2024 1:03 PM  
**To:** Weidong Peng (Credit) <[Weidong.Peng@vanquis.com](mailto:Weidong.Peng@vanquis.com)>  
**Subject:** Scorecard training docs?

Hi Weidong,

Hope your well 🙂

Rizwan suggested I approach you on this question: Do you have any internal training documentation on Scorecard Model Development I could look at please?

I'm doing an Apprenticeship for personal development. One of the things I'm doing for that is building a Scorecard within the Java language & as part of it I need to explain what one is to a non-technical stakeholder. My own knowledge admittedly isn't on the end-to-end Model Development so I was hoping we had some documentation around?

Kind Regards,

Benjamin Roberts

Lead Credit Risk Analyst – Credit Systems

Phone: 07798 848049

Email: [Benjamin.roberts@vanquis.com](mailto:Benjamin.roberts@vanquis.com)

Address: 1 Godwin Street | Bradford | West Yorkshire | BD1 2SU

Logo

Description automatically generated

Figure 24 - Email Chain with colleague for peer review

### Policy Rules

Policy rules check the available data on the applicant for specific signs of financial difficulty or any other indication that the applicant may be outside the credit institutions risk appetite.

Policy Rules can be of 1 of these types:

* **Decline** = If the logic is met, no offer of credit given to the applicant
* **Accept** = Valid only if the logic for all instances of the other 2 types are not met. Represents no concerns found on the provided data, an offer of credit can be given

|  |  |  |  |
| --- | --- | --- | --- |
| **Code** | **Rule Name** | **Decision** | **Logic** |
| D\_001 | Worst Arrears status is 2+ payments behind | Decline | IF DecisioningDataRow.E1B08 regex match 2|3|4|5|6|8 THEN FAIL |
| D\_002 | Experian Score below threshold | Decline | IF DecisioningDataRow.E5S051<= 600 THEN FAIL |
| D\_003 | Scorecard | Decline | IF (  decisionPath = “CHAMPION” AND Scorecards.Champion.score < 600  )  OR (  Scorecards.Challenger.score < 580  ) THEN FAIL |
| D\_004 | Unemployed | Decline | If EmploymentStatus == “Unemployed” |
| A\_001 | Accept | Accept | If none of the above Policy rules are hit |

# Testing the program

To be sure my program worked as intended and to the project requirements, testing was necessary. This was approached in 2 ways:

* Unit Tests via Junit 5
* End to end tests via running the program in the Eclipse IDE

## JUnit Unit Tests

When programming logic within the bespoke tool I use in my workspace, Unit Tests are used to ensure components of the code work as expected before a deployment is done. These are small scale tests that can be separated from the main program & re-used when any changes are made, allowing code quality to be maintained (AWS, n.d.). JUnit 5 is a Java package that provides a framework for writing these test & executing them (TutorialsPoint.com, n.d.). Examples written for this project are covered below:

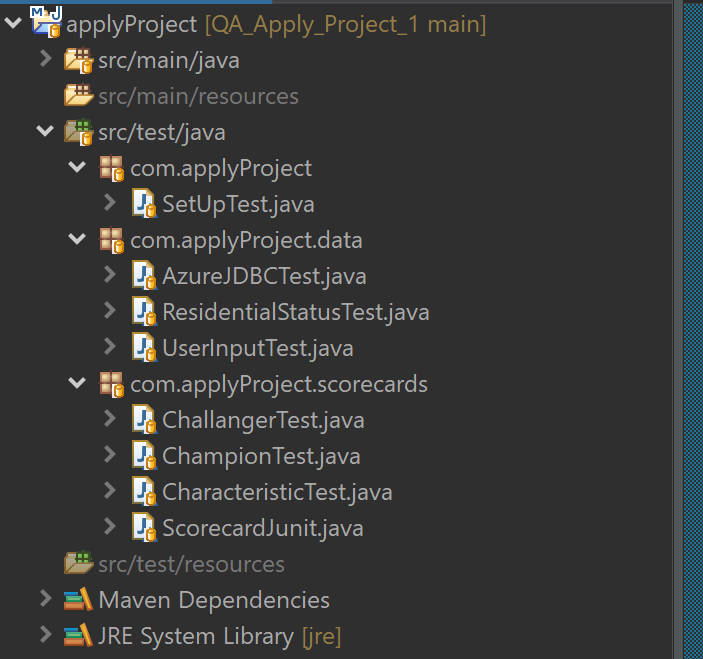


Figure 25 - JUnit 5 Test cases written to test individual parts of the program independently

A screen shot of a computer

Description automatically generated

Figure 26 - A test case written to test a program set up class that reads properties from an external file

## End to End

To fully test the program, it was executed within the Eclipse IDE:

A screenshot of a computer

Description automatically generated

Figure 27 - A demonstration of running the program within the Eclipse Software, with console output

Towards the end of the program, there are steps to output decisions made to the console. This includes the full Data Model used. This allowed manual review of the decisions made vs the specifications covered in this document.

A black screen with white text

Description automatically generated

Figure 28 - Console output for supporting debugging the program

## Debugging Errors

These forms of testing allowed me to identify unexpected behaviour (i.e. bugs) in my program. Below are 2 examples I’ve pulled out for demonstration.

### Probability of Default - Result always Infinity

One issue the Unit Tests on the Scorecards uncovered was that the Probability of Default function would always return an “Infinity” value, regardless of the formula input. This made the calculation effectively useless for it’s intended purpose.

To start debugging this, I added a specific check to my Unit Test to check for infinity values:

A screenshot of a computer program

Description automatically generated

Figure 29- Unit Test for testing probability of default

With some trial and error when running this test, I was able to correct the error (unfortunately I did not have a specific commit in my GitHub repository capturing this).

### Offers class returning exceptions

One Issue I faced with my program was that the Credit Offers the “Offers.java” class I wrote was not returning any of the expected results.

Examples of these tests are below:

A computer screen shot of code

Description automatically generated

Figure 30 - Unit Test on the Offers class

This test works by creating a new Scorecards & assigning a Score (via the constructor) and setting a Score Index of 1. The Score Index is what actually determines the maximum offer an applicant can receive e.g. in this test, we’re expecting the Index of 1 to mean that person could ask for up to £3000.

When I ran this test however, this was not what I was getting. Instead. The Fail trace should Illegal Argument Exceptions being raised:

A screenshot of a computer program

Description automatically generated

Figure 31- Unit Test results showing Illegal Argument Exceptions

After reviewing the code, I spotted I had neglected to add in the “break” statements for the Switch-Cases used within the “setMaxOffer” method this Unit Test was testing:

A screen shot of a computer program

Description automatically generated

Figure 32 - Java Switch case missing Break statements

The “break” statements are essential in Java’s implementation of the switch case, as they prevent Java from executing the next case (Geeks for Geeks – Switch, 2024). Without it, what happens here is that every switch case gets evaluated, even if a matching case is found. Ultimately, this ends up with the “default” case running, which generated the Illegal Argument Exception causing my unit test to fails. Once corrected, the test pass as expected:

A computer screen shot of a program

Description automatically generated A screenshot of a computer program

Description automatically generated

Figure 33- Correcting the Switch Case, allowing the tests to pass

Attendees (by initials):

* BR, TD [Head of Function, cover for RK]

Apologies (by initials):

* RK (on leave)

Positives

* Reusable code is shown via the Scorecard and Policy rule templates.
* JSON output of the decisions made provides a full view of the data used for auditing.
* Champion/Challenger scorecards considered and implemented logically.
* Some Error Handing steps in place to prevent redundant program execution if the program could not complete e.g. database credentials missing.

Possible Improvements

* Error Handling has room for improvement e.g. if the program is unable to connect to the database, a specific Policy Rule could be built for reporting.
* Scorecard points, index boundaries etc could be better placed as configurable values (ether via external config file or in the SQL database), which avoids hard-coding them into the complied program (and difficult to change)
  + Also creates possibilities to hand controls of this to business teams, potentially creating competitive advantage via fast to implement change.
* Database timeouts should be explicitly considered e.g. 30 second timeout.

Agreed overall the program met the Acceptance criteria set out and had scope to improve should the project be developed further.

Figure 34 - Meeting Minutes for presentation on final version with employer

Overall, I believe this went well. My employer agreed the project criteria has been met and also provided many valuable points of feedback from a less technical, more business practical point of view.

# Conclusions

## Project Outcome

Overall, the project succeeded in meeting the criteria set out in the introduction. The major functionality (Scorecards and Policy Rules) used within the bespoke Credit Decisioning software platform my team used has been developed into the program with re-usable classes that are suitable for creating many instances. A Credit decision is also made based on data retrieved from a backend database.

Additional Java Libraries were also discovered to expand the functionality of the program to effectively meet some of the requirements. For example, using the GSON library allows representations of the Java objects to be converted into JSON. That greatly simplified the requirement for retaining a record of the transaction, as this meant that all the data could be added to a Data Model object. All that was needed then was to save this Data Model to the logged history.

There was a significant amount of pre-requisite steps needed to get this project working e.g. setting up an Azure SQL database and loading data into it. This will be useful for later projects as a re-usable resource.

## Future Additions

Below are a few ideas I have in mind to expand this project once future content on the apprenticeship course is covered:

* Front end interface to collect more data on an applicant e.g:
  + Addresses
  + Income
  + Expenditure
* Submit the collected applicant data to the program as a semi-structured message body e.g. JSON
* Use previous runs of the application within business logic e.g. check the history for a count of same person re-running the program
* Replace the random database lookup with a connection to the Delphi Select Sandbox API
* Automate execution of the JUnit test cases & generate a report on the results

Reviewing with my employer also identified areas for improvement:

* Change approach to setting values to allow configurations like Scorecard points to be done outside the program, making them easier to change (potentially by business users as well)
* Improved Error Handing, in ways that made Management Information (MI) available to business users
  + Specifically, an explicitly defined timeout on the database connection

# References

AWS (n.d.). *What is Unit Testing? - Unit Testing Explained - AWS*. [online] Amazon Web Services, Inc. Available at: <https://aws.amazon.com/what-is/unit-testing/>.

Bionic Turtle. (2011). FRM: Logistic distribution maps credit score to probability of default (PD). [Online]. Youtube. Last Updated: 2011. Available at: <https://www.youtube.com/watch?v=HnQ6QjeGMm0> [Accessed 9 May 2024].

Data Protection Act. (2018). Data Protection Act 2018. [Online]. Data Protection Act 2018. Available at: https://www.legislation.gov.uk/ukpga/2018/12/contents/enacted [Accessed 17 June 2024].

Experian. (2024). DelphiSelect API Documentation - for Credit Decisioning data. [Online]. DelphiSelect API Documentation. Last Updated: 2024. Available at: <https://developer.experian.com/products/uk/delphi-select/delphi-select-api-docs> [Accessed 17 May 2024].

FICO. (2020). A Guide to Champion/Challenger Testing Approaches: Overview and Benefits. [Online]. FICO Community. Last Updated: 2020. Available at: <https://community.fico.com/s/blog-post/a5Q2E000000YNqoUAG/fico2227> [Accessed 17 May 2024]

Geeks for Geeks - Switch. (2024). SQL Injection. [Online]. Switch Statements in Java. Last Updated: April 2024. Available at: https://www.geeksforgeeks.org/switch-statement-in-java/ [Accessed 17 June 2024].

QA Apprenticeships. (2024). Module 1 - Java JDBC Slides. [Online]. QA Apprenticeships. Last Updated: 2024. Available at: <https://github.com/broberts-gtw/LVL_4_Software_Dev_Repo/blob/main/Module%201%20-%20Intro%20to%20Software%20Development/Live%20Course%201%20Materials/__Course%20Stretch%20files/Java/presentation/JDBC.pptx> [Accessed 1 May 2024].

Microsoft Azure. (2010). Adding Users to Your SQL Azure Database. [Online]. azure.microsoft.com. Last Updated: June 21, 2010. Available at: <https://azure.microsoft.com/en-us/blog/adding-users-to-your-sql-azure-database/> [Accessed 3 May 2024].

Microsoft. (2023). Northwind Database - TSQL. [Online]. Northwind Database - TSQL. Last Updated: 2023. Available at: <https://github.com/microsoft/sql-server-samples/blob/master/samples/databases/northwind-pubs/instnwn> [Accessed 1 May 2024].

Microsoft. (2024). Database-level roles. [Online]. Learn Microsoft. Last Updated: 2024. Available at: https://learn.microsoft.com/en-us/sql/relational-databases/security/authentication-access/database-l [Accessed 5 June 2024].

Snowflake. (2024). Access Control. [Online]. Snowflake. Last Updated: 2024. Available at: <https://docs.snowflake.com/en/user-guide/security-access-control-overview> [Accessed 5 June 2024].

Stack Overflow. (2008). Can an abstract class have a constructor?. [Online]. Stack Overflow. Last Updated: Updated Dec 30, 2016. Available at: <https://stackoverflow.com/questions/260666/can-an-abstract-class-have-a-constructor> [Accessed 3 May 2024].

Stack Overflow - ArrayList. (2021). Adding objects to an array list -. [Online]. Stack Overflow - ArrayList. Last Updated: 2021. Available at: <https://stackoverflow.com/questions/70406298/adding-objects-to-an-array-list-cannot-invoke-xxx-add-b> [Accessed 14 May 2024].

TutorialsPoint.com. (n.d.). JUnit - Test Framework - Tutorialspoint. [online] Available at: https://www.tutorialspoint.com/junit/junit\_test\_framework.htm.

W3Schools. (Unknown). Java Abstraction. [Online]. W3Schools. Available at: <https://www.w3schools.com/java/java_abstract.asp> [Accessed 3 May 2024].

## Java Dependencies

Bechtold et al. (2024). JUnit Jupiter API. [Online]. JUnit User Guide. Last Updated: 2024. Available at: <https://junit.org/junit5/docs/current/user-guide/> [Accessed 5 June 2024].

Google. (2024). GSON Libary. [Online]. Maven Central Repository. Available at: <https://mvnrepository.com/artifact/com.google.code.gson/gson> [Accessed 5 June 2024].

Microsoft. (2024). MS SQL JDBC. [Online]. Maven Central Repository. Last Updated: 2024. Available at: <https://mvnrepository.com/artifact/com.microsoft.sqlserver/mssql-jdbc> [Accessed 5 June 2024].

## Scorecards Theory & Probability of Default

Anderson, R. (2007). The Credit Scoring Toolkit: Theory and Practice for Retail Credit Risk Management and Decision Automation. Oxford University Press.

Huang and Scott. (2017). Credit Risk Scorecard Design, Validation and User Acceptance. [Online]. Credit Risk Scorecard Design, Validation and User Acceptance. Last Updated: 2017. Available at: <https://cer.business-school.ed.ac.uk/wp-content/uploads/sites/55/2017/03/Credit-Risk-Scorecard-Design-Huang-and-Scott.pdf> [Accessed 17 May 2024].

Mays, E. (2004). Credit Scoring for Risk Managers: The Handbook for Lenders. South-Western Educational Publishing.

Siddiqi, N. (2017). Credit Risk Scorecards: Developing and Implementing Intelligent Credit Scoring. Wiley.

Silva et al. (2020). A logistic regression model for consumer default risk. [Online]. National Library of Medicine. Last Updated: 2020. Available at: <https://www.ncbi.nlm.nih.gov/pmc/articles/PMC9041570/> [Accessed 9 May 2024].

Thomas, L. C., Edelman, D. B., & Crook, J. N. (2002). Credit Scoring and Its Applications. Society for Industrial and Applied Mathematics.