Jason Eide, Asif Zaman, Ethan Morris, Jose Hernandez

CSCE 3513 Software Engineering Spring 2015

Abstract

This document contains the report for the first iteration of a Personal Financial Management System

Project Deliverable

Iteration #1 Report

Group 8

This document gives an overview of the first iteration of the Personal Financial Management System. In this report, we first discuss the delivered features/user stories (specified by the customer). The next section goes over the methodology used in building this software system, specifically principles of Extreme Programming (XP) that were utilized. Finally we outline the cases used for testing each of the user stories.

**User Stories**

The customer requested for this iteration our primary focus should be on implementing a graphical user interface (GUI), plus a few (unspecified) user stories/modules. Regarding the user stories, we opted to implement the Real-Time Expense/Income Entry, the Savings Calculator, and the Debt Payoff Calculator. Learning how to implement the GUI was the lion’s share of person-hours worked for this iteration. Prior to this project, our team had virtually no experience in developing desktop applications with graphical user interfaces in Java. In the end, we implemented the user interface using the JavaFX GUI library. Once that framework was in place, we moved on to begin implementing the user-stories. The Real-Time Expense/Income Manager is the main component of the project, so that was a natural choice to include in this first iteration (in the actual product, the Expense/Income Manager is the “Transactions” tab). As the customer left it to us on which user-stories to implement, we then opted to include the Savings Calculator and the Debt Payoff Calculator. Initially the architecture design called for the Debt Payoff Calculator to be a secondary component branching off of the Savings Calculator. During the implementation stage however, we realized that the Debt Payoff Calculator should be a separate primary component, as a user may want to utilize those functions without tying it into the Savings Calculator. We explicitly choose not to implement the Budget Plan, Stock Simulator, or the Receipt Manager during this iteration. We needed the Expense/Income Manager to be almost fully operational first for the Budget Plan to function properly, as it relies on data from the main component. The Stock Simulator and Receipt Management components rely on the database (which is scheduled for the next iteration). For this iteration, our group put in a total of approximately 59 person-hours (Jason put in about 20 hours and Jose, Ethan, and Asif all put in about 13 hours each). We did not have an initial estimate to compare this too, as we were entirely unsure on how long it would take to learn and implement a GUI.

**Implementation of Extreme Programming Practices**

For this software system, we are employing Extreme Programming (XP) for development. This includes principles such as incremental planning, small releases, simple design, pair programming, collective ownership, sustainable pace, and on-site customer interaction.

1. **Incremental Planning** – Our requirements were created through the use of user stories, which were reviewed by the customer. The customer determined which stories should become development “tasks”.
2. **Small Releases** - The Expense/Income Manager is the main component and also provides the minimal useful set of functionality to the customer. Future releases will add functionality to the overall system.
3. **Simple Design** – Each individual component only executes the functionality for the user story that it solves.
4. **Pair Programming** – Approximately 40 of the 59 total person-hours on this iteration were through pair programming. Jason and Ethan worked as a pair for 9 hours on the Expense/Income Manager component (the main component) and Jose and Asif worked as a pair for 11 hours on the Savings Calculator and the Debt Payoff secondary components.
5. **Collective Ownership** – During our pair programming sessions, our group actually worked together as a whole. This allowed each programming pair access and insight on what methods and development tasks the other pair was utilizing. Additionally, decisions individual components or the overall project were made together as a group to solidify collective ownership. Finally, all code is available to all members of the group.
6. **Sustainable Pace** – Group members put in hours reflective of their overall workload which helped combat loss of code quality and production. Long programming sessions ended if productivity loss became noticeable.
7. **On-site Customer** – Early during this iteration cycle, the customer met directly with the entire team to discuss which system requirements should be implemented. Additionally, the customer is available for consultation at any time through email as well as multiple days per week in person (for this particular iteration, additional consultation was not necessary).

The biggest challenge for this iteration was implementing the GUI. Jason found and began researching the GUI library JavaFX a few days before our pair programming sessions. He spent quite a bit of time teaching the rest of our team how to use JavaFX before we could break off into pairs and work on components. Now that all four group members have experience building the GUI, work on future components (and improving the components of this iteration) should be able to progress at a faster rate.

**Testing**

As our software system requires input for each of the components of the first iteration, testing was mainly in the form of potential user input. In the tables below we outline the various test cases.

Table 1: Test Cases for Transactions Tab (Expense/Income Manager User Story)

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Functionality | Type | Value | Recurring | Expected Output | Matches Expected? |
| Add | Expense | 123.456 | No | -$123.46 | Yes |
| Add | Income | 27451.74 | No | $27,451.74 | Yes |
| Add | Expense | 1000 | Yes | -$1,000.00 | Yes (for current iteration) |
| Edit “Paycheck” | Income | 421.27 | Yes | $421.27 | Yes (for current iteration) |
| Edit “Electricity” | Expense | 107.3 | Yes | -$107.30 | Yes (for current iteration) |
| Delete | Expense | -7.50 | N/A | (Item line deleted) | Yes |
| Delete | Income | $367.48 | N/A | (Item line deleted) | Yes |

\*The output for recurring expenses and incomes will be slightly modified for future iterations to help differentiate from one-time expenses/income.

Please note that for the Transactions Tab, the date fields and item descriptions must also be filled out. For brevity’s sake on the table, we did not include these. However, please know that the expected output should match the date and description exactly, and for all test cases this was true. For the Transactions Tab, we also tested for non-numerical input in value fields as well as leaving fields blank or trying to edit or delete items without first selecting an item: for all of these cases, the program displays a dialog box with an error message and does not let the user proceed without fixing the error.

The table below displays the test cases for the Debt Tracker (Debt Payoff Calculator User Story). The expected value for each of the test cases is simply the user input displayed on the list with proper formatting for all variables. That is to see that monetary amounts are formatted with a dollar sign, proper comma placement, as well as precision up to 2 decimal places. The interest rate is formatted to precision up to 3 decimal places as well as a percentage (%) sign displayed after the number. Similarly to the Transactions Tab, test cases with negative, non-numerical, blank, and incorrect characters as the values were tested—all of the aforementioned test cases result in a dialog box directing the user to correct the error before being able to proceed. It is also important to note that the Calculate Payoff function of the Debt Tracker is not fully-operational yet; it will be implemented in the next iteration.

Table 2A: Test Cases for Debt Tracker (Debt Payoff Calculator User Story)

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Functionality | Name | Balance | Min Payment | Interest Rate | Matches Expected? |
| Add | Mortgage | 210341.91 | 1200 | 3 | Yes |
| Delete | Student Loan | $32,000 | $350.00 | 1.250% | Yes (Line item deleted) |
| Add | Credit Card | 837.8274234 | 100 | 27 | Yes |

Table 2A: Expected Values for Table2A Input

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Functionality | Name | Balance | Min Payment | Interest Rate |
| Add | Mortgage | $210,341.91 | $1,200 | 3.000% |
| Delete | N/A | N/A | N/A | N/A |
| Add | Credit Card | $837.83 | $100.00 | 27.000% |

The Savings Calculator user story was the final component of the system to test. This component calculates the number of months it will take you to save for your goal amount based on your monthly savings allotment. It will also allow you to add a recurring expense based on these numbers to your Expense/Income Manager (the Transactions tab). Once again we tested with specific numeric input, as well as the usual test cases that trigger the error message dialog boxes (non-numeric input, blank input, negative values, etc.). The table below reflects valid numerical input (as all non-valid tested triggers the error messages).

Table 3: Test Cases for Savings Calculator

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Functionality | Goal | Allotment | Expected Result | Matches Expected? |
| Add Transaction | 123654 | 1000 | Correctly add as recurring expense to transaction tab | Yes |
| Calculate | 1500 | 75 | Console output:  “It will take you 20 months to meet your saving goal of $1,500.00” | Yes |