The Design and Build of a Simple Personal Finance System, Focused on Budgeting and Expenditure Analysis

Word Count: *11,263* excluding appendices and images.

Claudius de Moura Brasil
BSc Computing Project Report
Birkbeck College, University of London

May 2018

Abstract

Personal finance systems exist in abundance nowadays, from open source to proprietary ones. They all tend to revolve around a basic common theme: providing accurate information about an individual's income and expenditure. Beyond this, they tend to vary in which features are implemented. The system designed and built for this project focuses on the use of the bookkeeping principle of double entry and the concept of pattern matching to find effective ways to cagetorise a user's expenditure, and provide them with relevant financial information to assist in decision making.

Contents

1 Introduction

A system could be summarised roughly as a solution to one or more problems. One of the first steps in order to build this kind of solution is to try to understand the problem – that is, try to map the requirements of the software. Vaasen et al. (2009 cited Boczko, 2012, p. 8) suggests that an accounting information system's main purpose is to provide information to internal and external stakeholders. Although this refers to accounting systems for businesses, it could be argued that this same definition can be employed to define personal finance systems – except that, in this case, the main stakeholder would be the individual using the system (that is, the user). In fact, one of the most widely known accounting systems available in the market, Quicken™, was conceived around the idea that there should be more efficient and less tedious ways to organise one's personal financial information than doing it manually (Quicken Inc., 2017). This project has been developed based on similar ideas.

It seems fair to infer that nowadays most of a user's financial transactions happen in ways that can be listed electronically (usually via their bank or credit card statements) – a study by Payments UK (2017), for example, indicates that there has been a rise in debit card payments over the past few years, and that the volumes of this type of transaction is likely to be higher than that of cash payments by the year 2021. Therefore, an assumption has been made that the users will require means of uploading a list of their financial transactions into the system in an electronic format.

The system created for this project intends to do just this. Its main feature, however, will be to allow the user to categorise expenditure based on patterns in the entries' descriptions. Aside from this, there will also be a feature to allow the user to view summaries of the income and expenditure over a period of time, and another one to generate budget forecasts for future periods based on the financial information already entered.

This report documents the work of the project. Each chapter delineates a specific aspect of the development life cycle, which is in line with the development process described in the following paragraphs. Chapter 2 outlines the identified requirements which were used as motivation for the system to be developed; the contents of Chapter 3 shows further analysis and concomitant design of the system and the solutions it brings; Chapter 4 outlines select aspects of the implementation stage which serve to emphasise the techniques used to implement the designed logic, or highlights areas where it was felt it was necessary to implement something slightly different that what was designed.

The system developed for this project has been modelled after the principle of *double entry bookkeeping*, from the accounting domain, which states that "money is never created or destroyed – it merely moves from one account to another" (Fowler, 1997, Section 6.2). More specifically, double entry is the principle which ensures that every transaction always affects two accounts, one being credited (Out) and one debited (In). An account, for the scope of this project, refers either to a category created by the user, or to the user's *cash book* – the contents of their bank account plus any manual entry which they make. In bookkeeping, each account can be classified as *asset, liability, income* or *expenditure*. Whether the account increases or decreases will depend on which of these categories it

falls under: *debits* will increase *assets* and *expense* accounts, and *credits* will increase *liability, capital* or *income* accounts (Wood et al., 2004, pp. 18-19).

Regarding the development method, an approach similar to that adopted by Bennett et al. (2010, p. 77) regarding software analysis an design has been employed, where no specific named methodology is espoused, but concepts of object-oriented analysis and design were applied, in an iterative and incremental fashion, using UML. More details about which concepts were used and the methodologies which originated them can be found in the following subsections.

The remaining definitions from Appendix I, including those of functional and non-functional requirements, will be employed when trying to classify the requirements and model the problem domain. The initial iterations will be focused more on the functional and usability requirements, paying some attention as well to specific non-functional requirements such as performance and security.

Throughout the report, the term *domain* will be employed, as defined by Evans (2004, p. 2), to define the "activity or interest of its user" – the "subject area to which the user applies the program".

Appendix III explains how to build and run the latest version.

2 Requirements

2.1 Business Case

Any personal accounting system should be able to provide accurate and relevant summaries of an individual's financial status. In order to do this, the user needs to be able to supply the system with the necessary data, so that it can be analysed and properly converted into knowledge.

The scope of the personal finance system developed for this project must include a feature to allow a user to upload their bank and credit card statements into it. Each transaction should be classified based on categories which the user will create. The user should then be able to visualise a summary of their income and expenditure by category and period, which should allow them to have a concise and clear visibility of how much they have earned and spent over any period of time. The category must be handled with care – there should not be a case where a user deletes a category and then all the entries in a period are lost with it, but if a user wants to change the name of a category they should be free to do so.

Since there may be other sources of income which the user may want to categorise (such as breakdowns of cash transactions from pocket money), a feature to allow for manual entries should also be made available. The user can declare a lump sum, and then break it down among categories. The option should allow them to choose whether the transaction is a credit or a debit from a specific category, and then provide the corresponding debit or credit to a category of their choice. For example, if the user withdraws £50, and spends half of it on weekly shopping and the other half on a cinema ticket, they should be able to 'credit' (withdraw) £50 from the *Bank* category, and then 'debit' £25 to both *Weekly Shopping* and *Entertainment* categories.

Once the system has enough data, it should be able to calculate a simple budget and display it for the user. The budget can be a simple average of income and expenditure over a long enough period of time, projected over the future month/year – it would only be used as a guideline for the user, anyway.

2.2 Functional Requirements

Based on the description above, a few functional requirements were identified. They are represented in the diagram on section 2.2.1 and the list, wireframes and activity diagrams on sections 2.2.2 and .

2.2.1 Use Case Diagram

Use case diagrams are UML constructs which were developed by Jacobson et al. (1992, cited Bennett et al., 2010, p. 154). The use case diagram on Figure 1 is used to illustrate the functional requirements identified for this project:

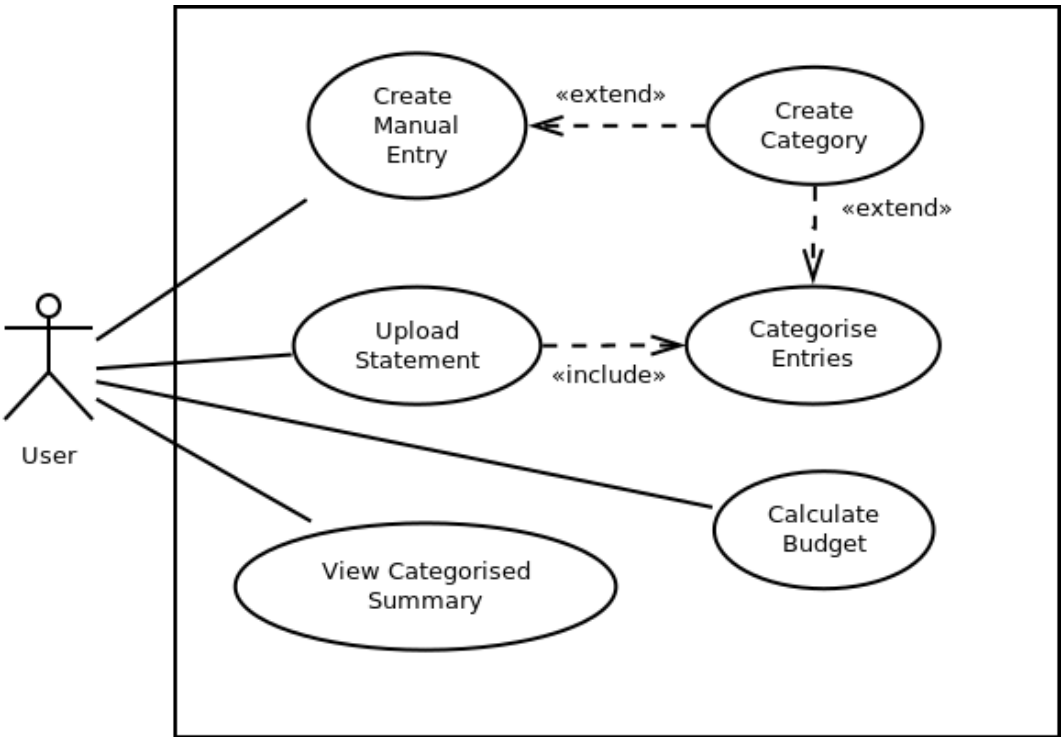

Figure 1: Use Case Diagram

2.2.2 Use Case List

Table 1 lists the descriptions for the use cases listed above:

Use Case	Description
Upload Statement	The user must be able to upload a list of their financial transactions, most likely their bank or credit card statements, in a valid format, and all entries should be categorised based on specific patterns *Includes*: Categorise Entries
Create Manual Entry	The user should be able to create a manual transaction for income or expenditure, include a date, amount and description, and classify it among existing categories or create new ones in the process *Extends*: Create Category
Visualise Categorised Summary	The user must be able to visualise a summary of their income and expenditure over a period of time
Calculate Budget	The user must be able to visualise a budget for future periods based on their income and expenditure data already entered
Categorise Items	Analyse the current entry and assign it to a category *Extends*: Create Category
Create Category	Creates a new category with the name suggested by the user

Table 1: Use Case Descriptions

The *Estimate Tax* feature was not included in these requirements, as the time constraints would not allowed for it to be implemented, but its specifications can be seen in Appendix II.

2.2.3 Designing the User Experience with Wireframes

The wireframe below (Figure 2) was created to better illustrate the *Manual Entry* requirement from the point of view of the user. It shows an example of an entry for a laptop and a licence for a proprietary operating system, which can then be broken down among different categories. The user has the option to use the percentage or the amount boxes in order to provide a breakdown, and they can also add new lines if more than one is required – the example shows two lines, but the default would be one. Under the category search box, if the user types a category name that does not exist they will be asked if they want to create a new one:

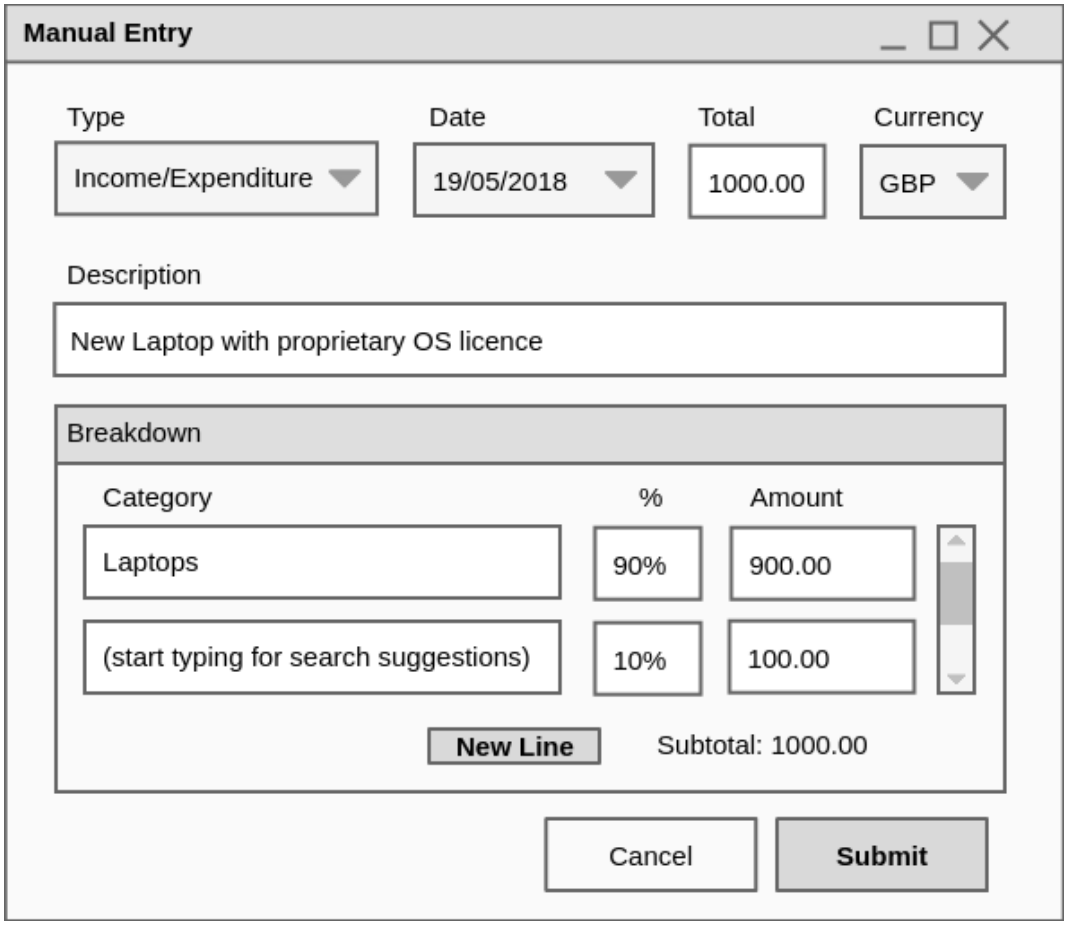

Figure 2: User interface wireframe for *Create Manual Entry* use case

And in order to better understand the relationship between *Upload Statement* and *Categorise Entries*, the activity diagram below (Figure 3) was developed:

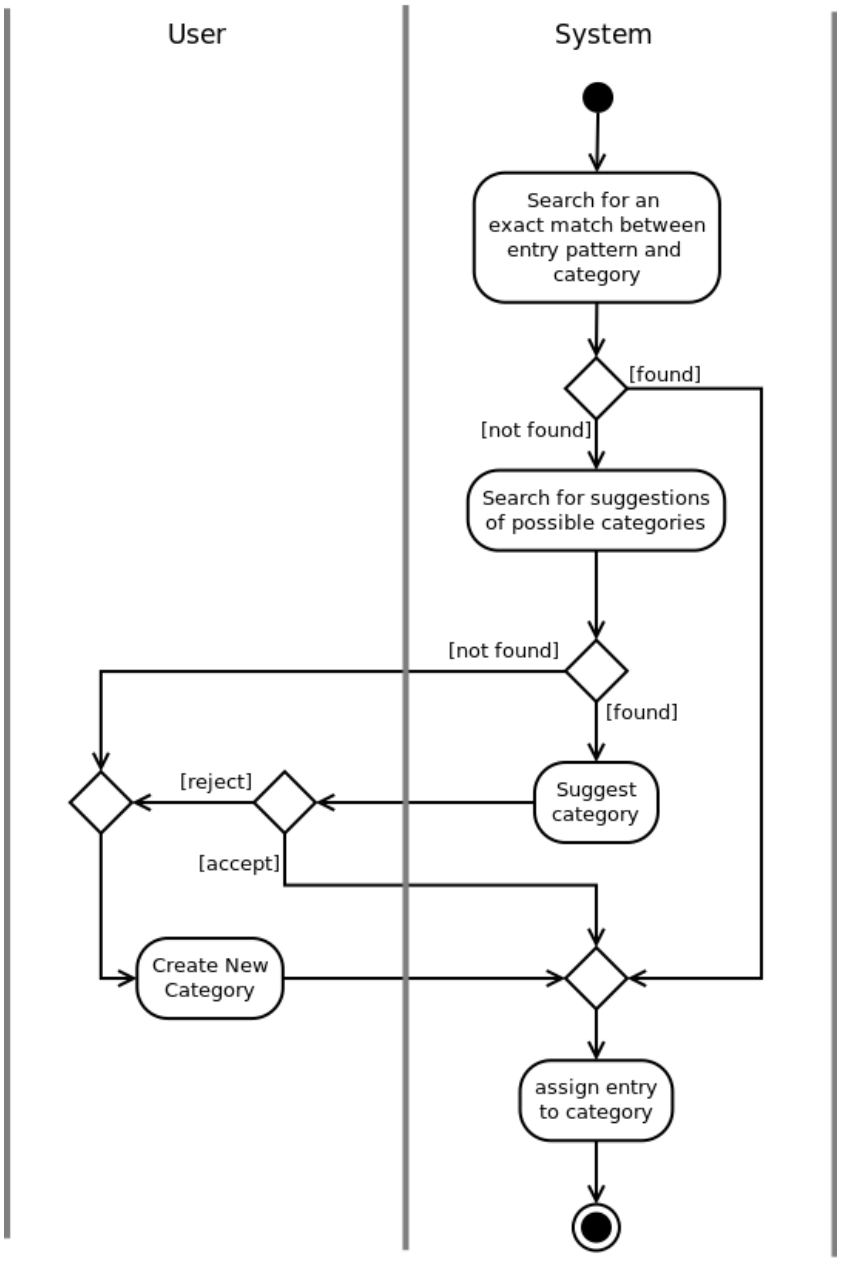

Figure 3

At the first few iterations, in order to provide a minimum viable product, the process of categorisation will be a blocking one consisting of multiple calls being made to the process above – for each call, the process will block awaiting user input when a category is not found. However, there are plans for future iterations where this process could be optimised by concurrency, and if there is enough time a more appropriate interface will be built for such a case.

The initial interface for uploading a statement should be a simple one, such as the one below (4), and at least initially the interface for manual entry will be used whenever user input is required:

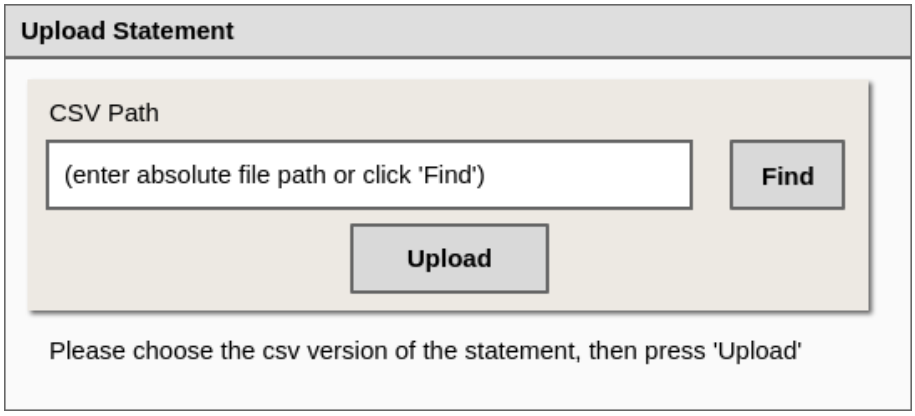

Figure 4

Below (Figure 5) is also a wireframe illustrating the GUI for *Visualise Categorised Summary*. The user should have an option to select the dates and, if they only want to see one category, the category itself:

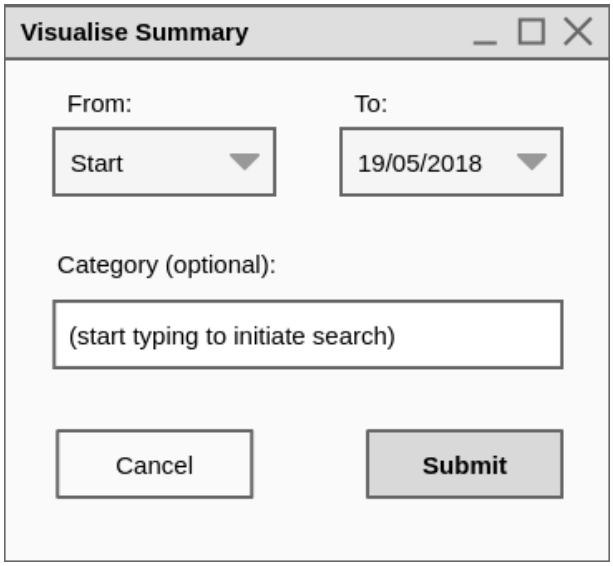

Figure 5

The dates field should allow them both to type a date or choose it from a drop down calendar. If no values are entered, the system will return a summary of all categories on the system.

2.3 Non-Functional Requirements

At this iteration, no significant non-functional requirements were identified, so they were not included in this report.

2.4 Requirements Capture Methods

The closest match identified to the techniques utilised for requirements capture for this project was *'Knowledge Acquisition'*, this relates to the process of capturing knowledge from an expert (Bennett et al., 2010, p. 150). In this particular case, though perhaps not qualifying as an expert, the author's own qualifications and experience in accounting and bookkeeping were used to generate the ideas from which the requirements were extracted.

Differently from the original plan, and more in line with the *incremental model* of development exemplified by Dawson, 2009, pp. 120-124, the requirements were gathered in full before any actual development started.

3 Analysis and Design

As described by Bennett et al (2010, p. 348), "in projects that follow an iterative life cycle, design is not such a clear-cut stage, but rather an activity that will be carried out on the evolving model of the system". Seeing that the development method being followed in this project is based on what should be an iterative approach, it was decided that the analysis and design of it would be done concurrently.

The system will be designed in a layered architecture, and will consist of three layers: *presentation*, *business Logic* and persistence. The presentation layer will take care of the vast majority of the interactions between the user and the system, and will depend only on the Business Logic layer below it. The business logic will contain the model of the concepts which will attempt to implement a solution to the problems outlined in the requirements, and will depend only on the persistence layer below it. Lastly, the persistence layer will ensure that the data is not lost when the application is stopped (Bauer et al., 2016, p. 32-33). This approach is also reminiscent of what has been described by Holmes (2016, p. 3) as full stack development for an application using a Model-View-Controller (MVC) architectural pattern, with one of the differences being that the latter provides more details on how and when the *view* should be updated.

Regarding the modelling of the domain, Fowler (1997, Section 1.3) defines a pattern as "an idea that has been useful in one context and will probably be useful in others". So, analysis patterns will be used "when trying to understand the problem" domain (Fowler, 1997, Section 1.1) in order to test this concept, and as an attempt to make use of the experience already acquired in the domain (or domains) in question. As emphasised by Bennett et al. (2010, p. 252), "a pattern is useful when it captures the essence of a problem and a possible solution, without being too prescriptive". So it may be the case that some of the patterns will be modified where necessary to optimally solve a problem.

Essentially, an analysis pattern consists of a structure of classes and associations which occurs often in many modelling situations related to specific domains (Bennett et al., 2010, p. 254).

Most of the aesthetic designs of the presentation layer happened in chapter 2.2, so this one will focus on the analysis and design of the business logic and persistence ones.

3.1 Business Logic

Seeing that this is a system dealing with finance, it would make sense to treat the categories as if they were treated as *accounts*. So, in order to make sure to imbue this system with knowledge acquired by more experienced programmers, analysis and design patterns will be employed.

It is also useful at this point to make a distinction between the types of classes used to model the domain between three possible kinds: the first are the classes which model

the interaction between the system and its actors – these are called *boundary classes*; the second kind are those classes which model information and/or behaviour or some concept or phenomenon – these will be called *entity classes*; and lastly, there are those classes which model transactions, coordination, control and sequencing of other objects – which are known as *control classes* (Jacobson et al., 1999, cited Bennett et al., 2010, pp. 198-201).

The first analysis patterns which seem appropriate are a modified version of the *Account* pattern, used to create the `Category` entity class, and the *Quantity* pattern for the `Amount` entity class (Fowler, 1997, Sections 6.1 & 3.1):

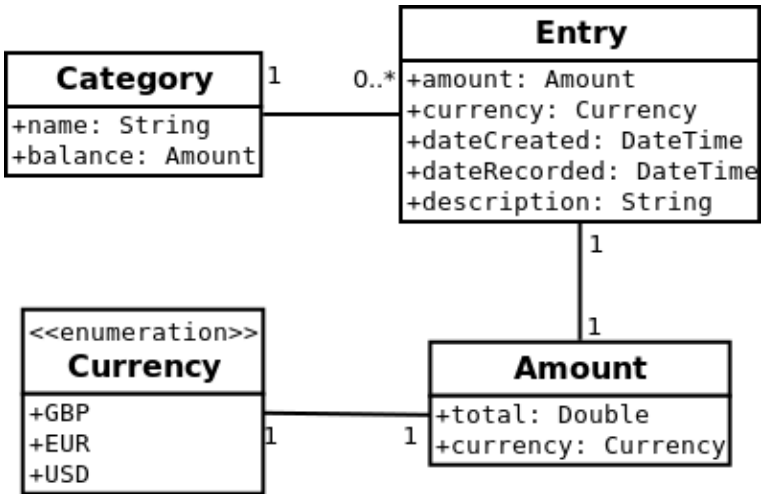

Figure 6

As implied by the diagram above, the `Category` class will be associated with instances of the `Entry` class. This is done so that the only way to change the total of a category is by adding positive or negative entries to it – for example, to indicate a credit to a category, a negative entry can be added to it. One of the modifications to the original Account pattern consists of the fact that, whereas in the original pattern an instance of `Account` would keep track of the balance, there is no need to keep track of the current balance in each `Category` – the purpose of the system is to allow the user to view a summary of income/expenditure by period, so this will have to be calculated each time.

Another design choice which can be observed in Figure 6 is that the `Amount` class also possesses an attribute for currency. This has been designed so as to allow for the possibility of extending the system to keep track of transactions in multiple currencies, although this was not a specific requirements. Initially, there will only be a single default currency (GBP), so this will be hardcoded. But the class should be implemented in a way which also allows for different currencies to be loaded from another layer, such as the database, or an external API.

The next step is to provide a way for these entries to be added to categories. For this to happen, there needs to be a constraint to ensure that double entry happens every time a change needs to be made to a category. One of the ways to achieve this is to apply the *Transaction* pattern (Fowler, 1997, Section 6.2):

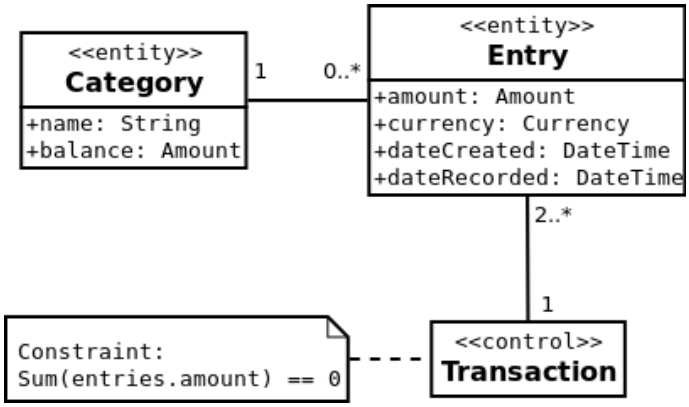

Figure 7

After having determined some of the analysis patterns which shall be employed, it makes sense to dive into a more specific analysis of the use cases described in Chapter 2. At this point the objective will be to start modelling classes and interactions based on concepts or things found in the problem domain. This will be done in the following subsections.

3.1.1 Create Manual Entry

The *Create Manual Entry* use case, which allows a user to input financial transactions individually using a specific interface, can be modelled as follows (Figure 9):

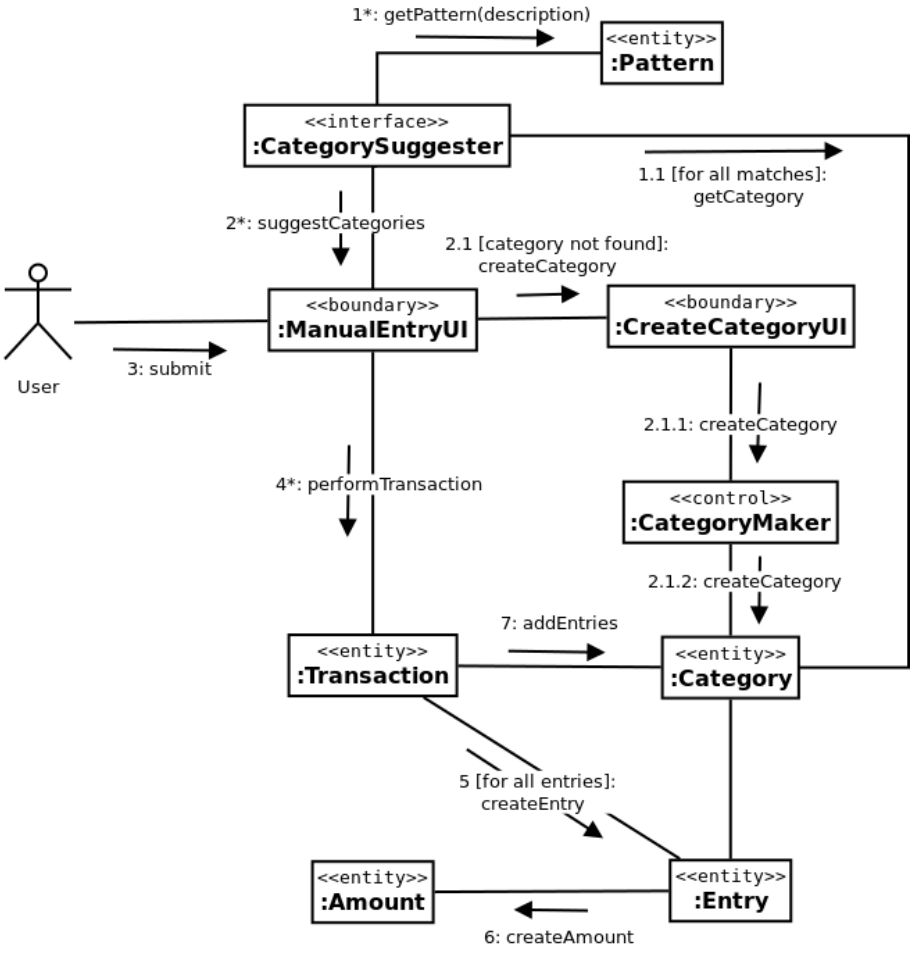

Figure 8: Communication Diagram for the *Create Manual Entry* use case

As the diagram above indicates, the `Transaction` class is responsible for the creation of new instances of `Entry` and `Amount`, which then get assigned to the `Category` instances chosen, or created, by the user. Once the user starts typing, the `CategorySuggester` is triggered to suggest categories. The actual implementation of how this suggestions happen may vary, but initially it should at least be based on what the user types in the search box. If the user chooses to create a new category instead, then they will be taken to the appropriate interface to allow them to do so. Once the user is satisfied and chooses to submit, the system will start to input the transactions in the appropriate category or categories.

Although `Transaction` was originally thought of as a *control class*, it was later decided that it would be beneficial to save the transaction information, so it was made into an *entity class* instead. It is also worth noting that, although implicit in the diagram and the wireframe on Figure 2, it is estimated that the user will start the interface, enter the transaction's details, and then start the process to find a category. Seeing that there will be a search suggestion at this point, it felt it was appropriate to have the diagram on Figure 9 have its first message sent at this stage.

It is also important to emphasise the fact that `CategorySuggester` is only an interface at this point, and that the suggester in the diagram will be any object which

implements this interface. This is to allow more flexibility in the implementation of the classes responsible for suggesting categories to the user.

3.1.2 Upload Statement

The user should be able to upload their bank statements, provided that they are in a suitable format. The specifics of the format will be described in the implementation phase, together with more information on how to encapsulate as much as possible the complexities related to the formatting of this information. For the analysis and design phases, the emphasis will be on modelling the objects and their interactions. The diagram (Figure 9) below illustrates this process:

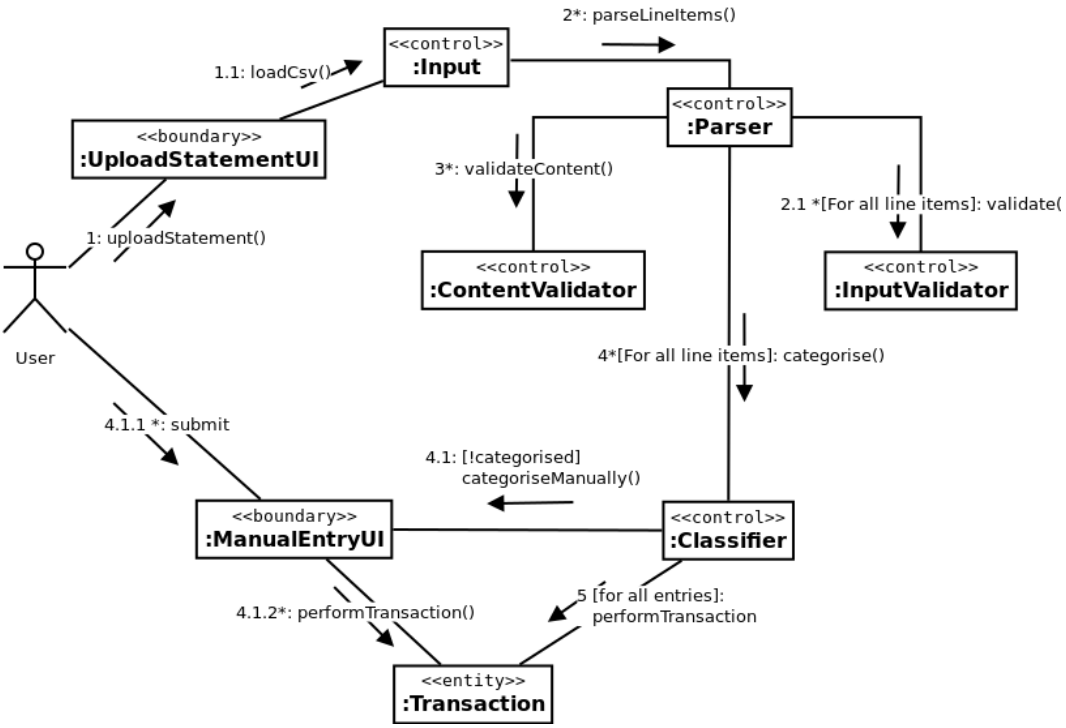

Figure 9: Communication Diagram for the *Upload Statement* use case

As can be seen in the diagram above, the process is spread among many classes. After the user uploads the statement, the loaded raw input will be sent to a `Parser` which will separate it according to the columns into the appropriate fields and line items. Then, what is now a collection of statement line items will be passed on to an `InputValidator` to make sure the user input is valid. Lastly, the resulting validated entries will be sent down to a `Classifier`, which will signal the relevant `Transaction` instance(s) to add the entries to their relevant `Category` instances – this last part has already been illustrated in Figure 9.

When the `Classifier` cannot match a line item against any of the existing categories, it will pass the line in question to the `ManualEntryUI`, which, apart from having all transaction details already populated, will rely on the process described at subsection

3.1.1 to properly categorise the item and then forward it to `Transaction` to create the categories.

3.1.3 Visualise Categorised Summary

As mentioned before, this feature will allow the user to view a summary of their income and expenditure by category. The user will enter the period which they want to examine, and the system will retrieve the categories which have entries with those dates. The system should then sort the categories by income or expenditure and by total, and then display it to the user. Optionally, the user can filter the output further by choosing a single category.

Below (Figure 10) is a communication diagram to illustrate this interaction:

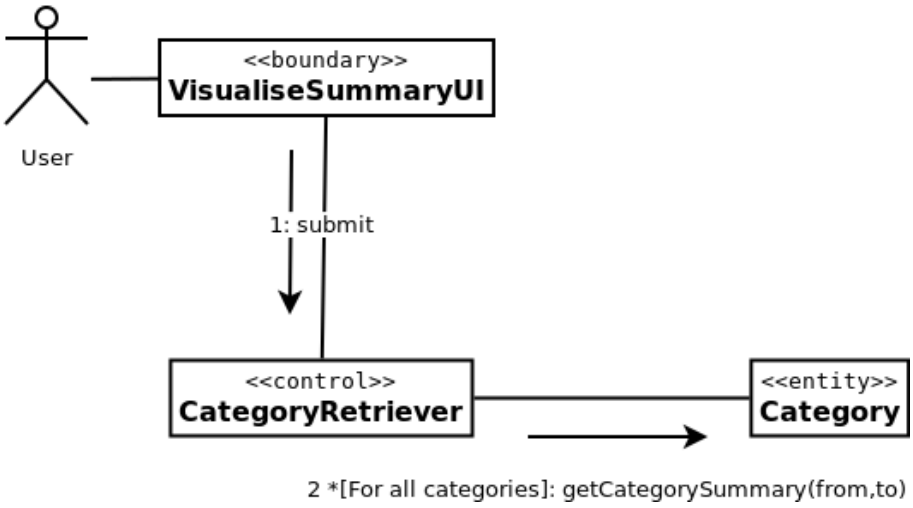

Figure 10: communication diagram for *Visualise Categorised Summary* use case

3.1.4 Calculate Budget

This feature will allow the user to request the system to calculate a budget for them over a period of time. What the system will do then is retrieve all categories over the last 12 months, then calculate their means and take them as ratio of the income over the same last period. Then, it will take the first few which make up more than 80% of the total, and add all the others which make up the remainder and add them up as 'other' or something similar. It will then show these totals to the user, in a similar way as it shows the categorised summary. The communication diagram below (Figure 11) illustrates the steps taken by the application layer.

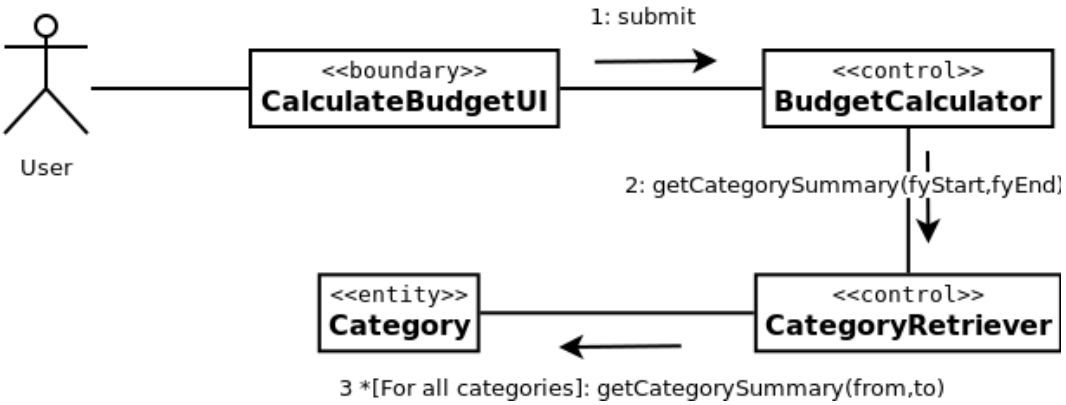

Figure 11: communication diagram for *Calculate Budget*

3.1.5 Final Class Diagram

The class diagram on Figure 13 is the result of joining the more relevant entities listed so far:

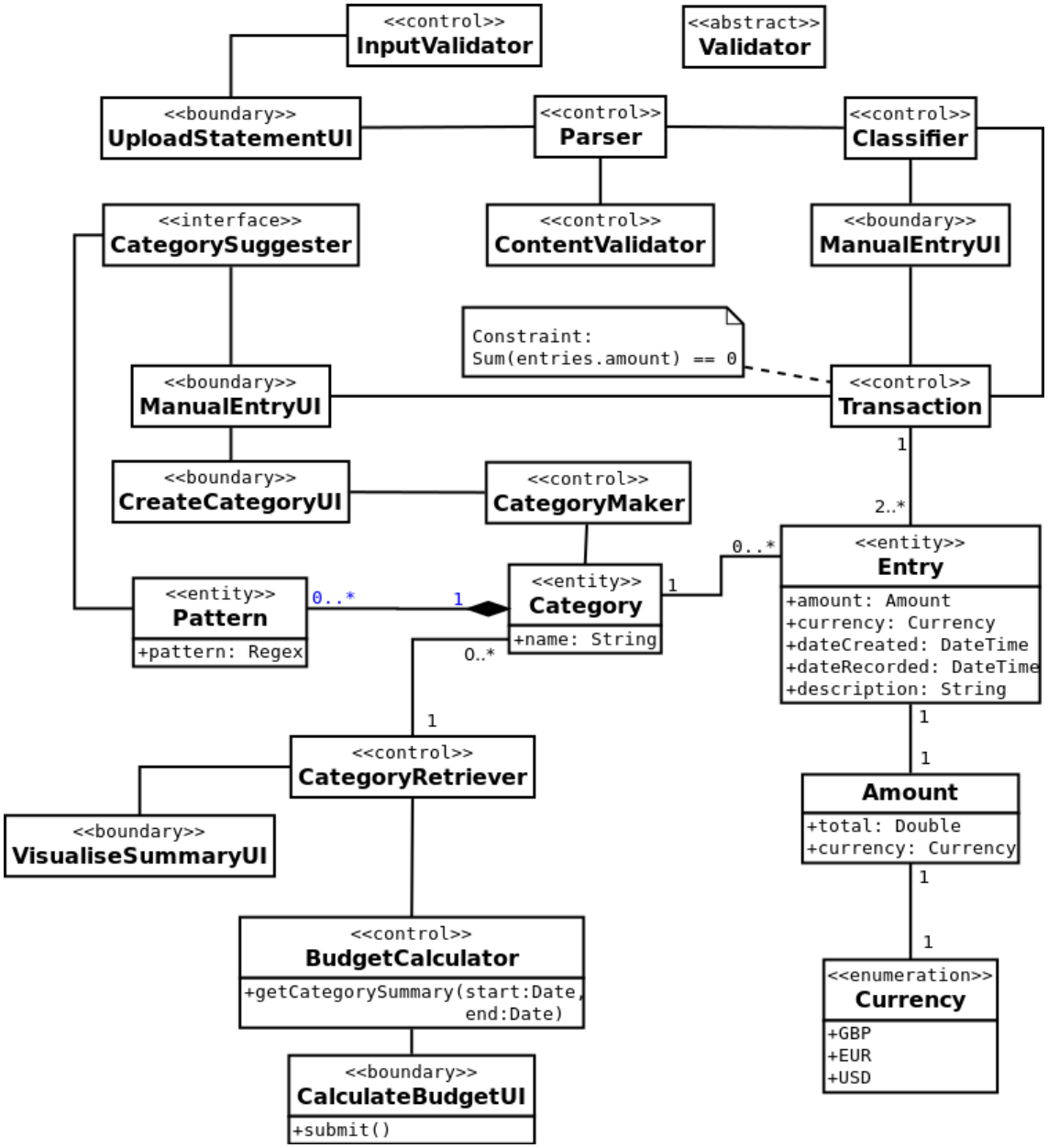

Figure 12: Class diagram of the most relevant classes of the Business Logic Layer

Manual entries and uploaded statements will inevitably always go against a Bank category. For this reason, there does not seem to be any harm in hard-coding the `Bank` category. In this case, since most transactions are going to affect it, it was decided that no patterns should be assigned to it. Due to this, it was decided to allow multiplicity of `0..*` to the association between `Pattern` and `Category`. This relationship has also been made into a composition, since patterns should not exist without categories on the system, but a category can exist without a pattern.

3.2 The Persistence Layer

This is the layer which communicates directly with the database. It tries to encapsulate as much as possible the details of how it accesses the outside world by only exposing one singleton object, the `PersistenceMediator`. This is the object responsible for manipulating the database, with the help of other encapsulated package members. It uses a *Java Properties* file to choose which database engine to use (at the moment only *MySql* and *H2* are available), but

3.3 Database Diagram

The UML diagram below shows the design of the database schema for the application:

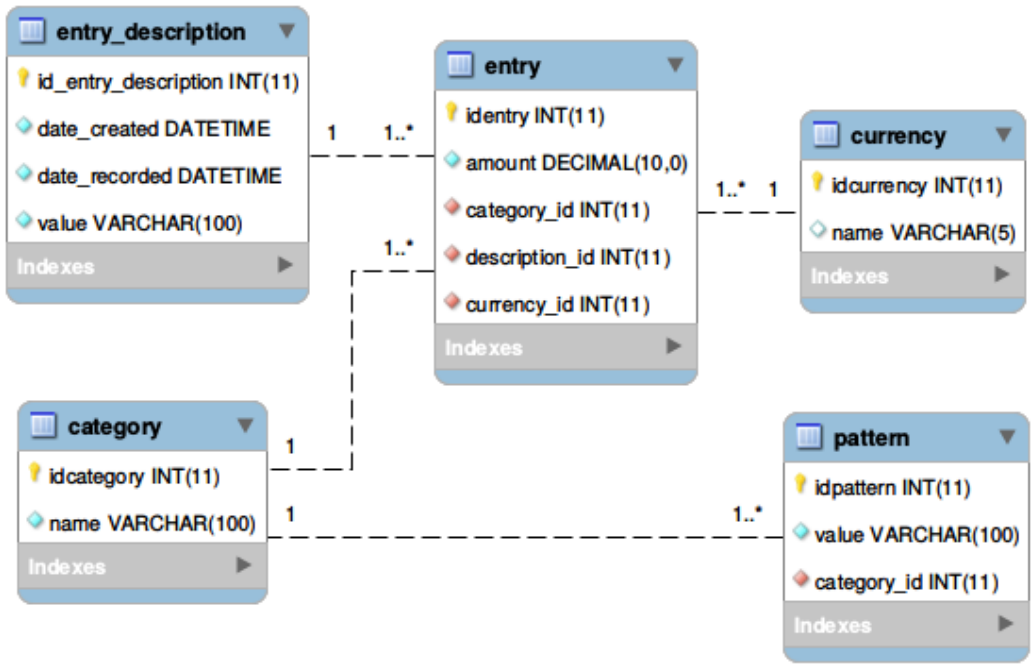

Figure 13: UML Diagram of database schema

As the diagram indicates, the entities and relationships of the database are very similar to those of the business logic layer, except that it was felt that, since due to double-entry the `Entry` instances will occur multiple time, the description should be given a table of its own in order to contribute to optimising storage space used.

One of the main reasons for choosing to model the application logic first, and then model the database, was that it would be more likely to ensure that the model (data structure) would reflect the needs of the application more accurately, as stated by Holmes (2018, p. 141). Also according to Holmes, the risk of having the model of the data adversely affect how the application will work and behave can be higher if the application design starts with the data model.

4 Implementation

The initial Minimum Viable Product (MVP) was implemented with major differences from the design outlined in chapter 3. The most visible one is the fact that (what is mainly) the *Mediator* pattern (Nikolov, 2016, Ch. 9, Location 3594) has been used in order to implement the dialogue between the presentation and other layers using a feature which resembles the MVC architecture, as mentioned in chapter 3. Normally, this type of architecture is implemented in order to make sure that multiple views can refer to the same core functionality (or model) without actually implementing that functionality themselves, which allows the view to be responsible for presentation and the model for liaising with the business logic through a controller. This also allows the controller to update multiple existing views based on what input was received from one view (Bennett et al., 2010, p. 381). The difference in the implementation for this project is that this last feature, essentially, was not implemented, since only one view exists at one time.

It is noticeable that the interface for manual entry available is different than that shown in Figure 2 is not exactly the one implemented: the feature to add a new line to the breakdown is missing, which makes the percentage of income irrelevant. This was once again due to time constraints, and will have to perhaps be done at a further iteration. Another difference is the fact that there was not enough time to implement a mechanism to suggest categories to the end user, so this will have to be left for a future iteration, outside of the scope of this report.

In the implementation phase of this project, the components were segregated into packages (each of which represents one of the layers mentioned in Chapter 3) and sub-packages. Each sub-package has been implemented so that it does not have any knowledge of the super-package, but rather exposes and depends on interfaces which exist within itself. The super-packages, then, implement the interfaces of the sub-packages, which allows them to interact with the sub-package components. The *Mediator* pattern was implemented at the default package level by the *InteractionMediator* and *PersistenceMediator* objects, both of which also happen to represent Scala's native implementation of the *Singleton* pattern (Nikolov, 2016, Ch. 6, Location. 2242). The following subsections will give more detail into the inner workings of the application and the techniques used to build it.

4.1 Presentation Layer

In the presentation layer, for this MVP, the *Scala Swing* package was used to design the *view*. The package itself consists of Scala wrappers for the *Java Swing* package, and one of the reasons why it was chosen was that, as with many other GUI packages, it already comes with an implementation of the *Observer Design Pattern* (Gamma et al., 1995, p. 293) in its capacity to react to events (Maier, 2009, p. 5; Nikolov, 2016, Ch. 9, Location 3731).

The entry point of the application is the `App` object, in the main package. This application works as a main method, and its only task is to call the `InteractionMediator.startup` method, which will send a message through the `PresentationMediator` instance returned

by the `PresentationBuilder` to start the view.

In its current implementation, the view is started by the `MainWindow` class, which extends the `scala.swing.MainFrame` class. A `MainFrame` is a Swing `Frame` – "a window containing arbitrary data" (Odersky et al., 2016, Ch. 34, Section 34.1) – that can send a signal to terminate the application when closed. `MainWindow` also has implementations of other methods to allow the application to close gracefully, all of which were inspired by the `SimpleSwingApplication` of the scala package (Maier, 2009, p. 2 & 3). The author chose not to make the first extend the latter was because it would have inherited the main method of the super-class, which, considering that it would be another entry point to the application, could cause confusion when trying to run the application (not to mention the fact that this is not the responsibility of the `MainWindow` class).

As mentioned in the beginning of this chapter, each sub-package has been designed so that it does not have to depend on the packages which envelop it. The diagram below (Figure 14) illustrates this using the interaction between the `InteractionMediator`, `SwingAmbassador` and the `presentation.swing` package and sub-packages as an example:

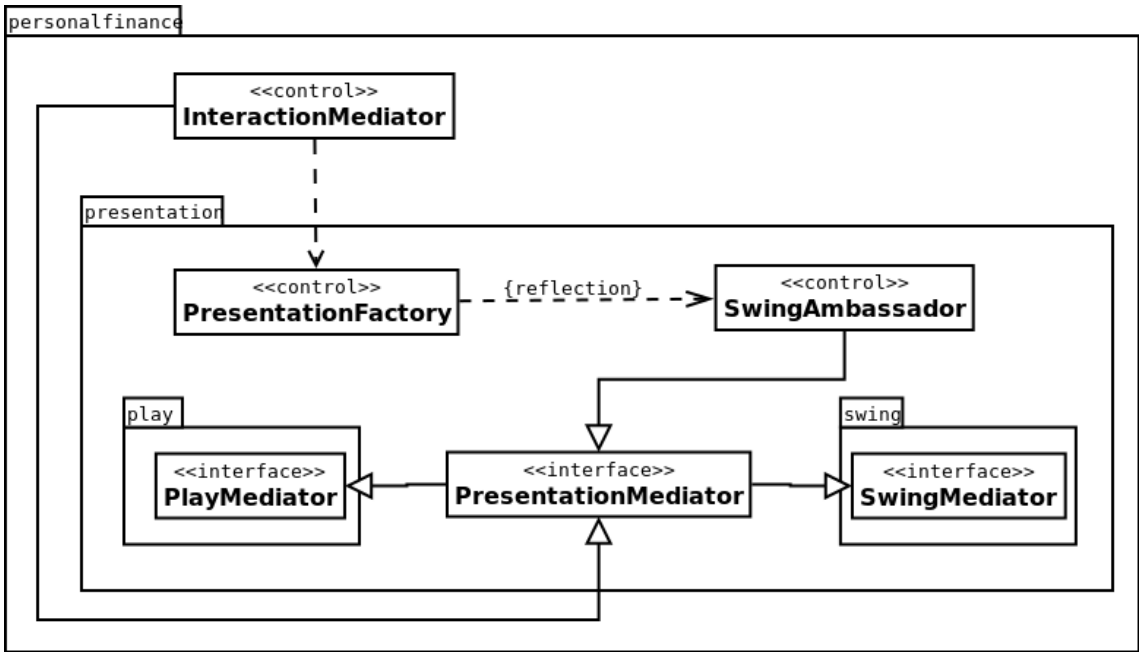

Figure 14

In order to preserve encapsulation, specifically for the `swing` package, only the `MainWindow` controller and `SwingMediator` interface are exposed to the outside world, with the first having a constructor dependency on the latter. This means that, in order to interact with the components of the package, any external classes will need to implement the `SwingMediator` interface. This is also useful if the `presentation.swing` package is to be extracted and used as a library in another project, and also leaves room for the *Adaptor* pattern (Gamma et al., 1995, p. 139), if it needs to interact with other objects which do not necessarily implement the interface upon which it depends.

One feature also noticeable in Figure 14 is the fact the `personalfinance` package is unaware of which of the view's implementation it is interacting with. The modified version of *Builder* pattern (Løkke, 2009, pp. 37-38) implemented on `PresentationBuilder`, which uses reflection to determine what implementation to build, allows for the view itself to be decided with the use of an external `.properties` file. This allows for the actual implementation which will be delivered by the Factory to change without the code having to be recompiled, which introduces flexibility to the application. The `play` package is only a placeholder at this point, as due to time constraints it was not possible to actually implement it, but it is there to exemplify how the project can be further extended.

Another element not seen in the diagram but which is present in the code is the fact that the dependency of the `MainWindow` class on the `SwingMediator` interface exists in the constructor, as seen in the code listing below:

```
class MainWindow(mediator: SwingMediator) extends Reactor {
```

This is an attempt at implementing the principle of *Dependency Injection* through *Constructor Injection* (Fowler, 2004). In the case of this project, the advantage of this simple piece of code is tremendous, especially because the fact that the `MainWindow` class does not have to instantiate a mediator – it simply relies on the fields published by the `SwingMediator` trait. Likewise, the fact that the `InteractionMediator` singleton object has a dependency on the `PresentationBuilder` only, allows for the actual implementation of the view to change without it having to change.

The approach described above has an inherent risk due to Scala's *mixin* composition. Essentially, since trait `PresentationMediator` inherits from multiple traits, any future extension to implement a new view would have to keep in mind that `PresentationMediator` will be implementing the behaviour of the last trait it implements in the source code. So any refactoring of this trait which extends other traits may generate unexpected behaviour, if this is not taken into consideration.

4.2 Business Logic

The business logic layer is the one which adhered more firmly to the model designed in section 3.1.5. The following are a few highlights from the implementation which are worth mentioning.

4.2.1 Scala Case Classes and the Prototype Design Pattern

One of the benefits of Scala are its case classes. Not just are they very useful for pattern matching, but also come with a few perks such as the `copy` method. This method allows for an object to be copied with some or all its members modified. It could be said that it is a language-native implementation of the *Prototype Design Pattern* (Nikolov, 2016, Ch. 6, Location 2461), and throughout the implementation and testing stage it proved to be a useful asset.

One of its many utilisations can be seen in the `Transaction` class, as one of the tools used to add entries to a `Category` without having to change the state of a specific instance – more similar to what is done in the *Functional Programming* paradigm:

Listing 2: extract of the `Transaction` class showing the `copy()` method in action

```
private def addEntries(tu: TransactionUnit): Category = {
  val cat: Category = tu.category
  val ents: Seq[Entry] = tu.entries
  if (cat.entries.isEmpty) cat.copy(entries = ents)
  else cat.copy(entries = cat.entries ++ ents)
```

In order to be concise where possible, where it was felt that the domain's requirements were captured accurately and was estimated that they are not likely to change, high level modules are depending directly on classes rather than interfaces. This can be seen as being in direct violation of the *Dependency Inversion* principle of SOLID (Martin, 1996), which some would consider to be a "risky"' move (after all, it is not always possible to predict which requirements will and which will not change), but others would believe that this was acceptable for more static requirements. The author decided to follow the latter.

However, this principle was still employed where it seemed beneficial to follow it – for example, in areas where it was felt that the implementation of certain algorithms could be further optimised in future iterations. This is the case with `transaction.RegexDateStringParser`: this class implements the `transaction.dates.DateStringParser` trait, upon which the higher level modules which use dates will depend. This is an example of one of the places in the code where the *Dependency Inversion* principle can be seen, but it is not the only one.

4.2.2 The engine behind it all: the `Classifier`

Another example of *Dependency Inversion* (though not of *Dependency Injection*) is the `Classifier` trait, and the class implementing it: `StringClassifier`. The `InteractionMediator` declares a variable of the first type, and then instantiates it with the latter. Therefore, although it still depends on the abstraction, it also has a dependency on the implementation, which would mean that in order to change it a slight refactoring would be necessary.

For this implementation, `StringClassifier` was built using longest matching prefix to match entries to categories: patterns are matched to the start of the description of every entry, starting from longest pattern to the shortest. This is so that if there is, for example, a pattern "Laptop for parent" which would match to category "Gifts", and another pattern "laptop", for category "personal equipment", then entries with the description "Laptop for Girlfriend" would not be picked up by the shorter match for laptop. Below is a code listing which shows how this was implemented:

Listing 3: Extract of the `StringClassifier` class, showing the algorithm currently used to match Entry descriptions' prefixes against patterns

```
private def classifyByDescription(categories: Seq[Category],
  entries: Seq[Entry]): (Seq[TransactionUnit], Seq[Entry]) = {
```

```
    // extract all the patterns from categories and sort
    // them by length, from longest to shortest
    val patternIndex: Seq[String] = categories
      .flatMap(c=>{c.patterns.list.map( _.value )})
      .sortBy(_.length)
      .reverse

    // match the full pattern against the prefix of every
    // entry description
    entries.map({
      entry =>
        (entry,patternIndex.find(
          entry.description.startsWith)) match {

        // if it finds them, return a transaction unit for
        // that category
        case (e: Entry, Some(pat)) =>
          TransactionUnit(categories.find(cat => {
            cat.patterns.list.foldLeft(false)({
              (test,catPat) =>
                catPat
                  .value
                  .toLowerCase() == pat.toLowerCase() || test
            })
          }).get,Seq(e))

        // if not, return the entry as uncategorised
        case (e: Entry, None) => e
      }
    }).partition(_.isInstanceOf[TransactionUnit])
      .asInstanceOf[(Seq[TransactionUnit],Seq[Entry])]
  }
```

As mentioned on the code listing above, the *longest matching prefix* is achieved by sorting all the patterns by length, from longest to shortest, then trying to match them against the prefix of the entry description in the same order. This way, even if an entry would be a match against more than one category, the pattern with the longest matching prefix will still be the first match.

One other interesting feature which can be seen in the above listing is how much is being done using the native `flatMap` and `foldLeft` methods of Scala collections. The `map` function and its derivates are normally implemented by types which can be classified as *Functors* in scala. Functors are challenging to explain concisely, but for this report it should suffice to say that they are *classes which implement the* `map` *method and conform to a set of laws called* **functor laws**. The `foldLeft` method of the collections are also an indication that they can be classified as *monoids*, which are another construct brought over from mathematics. The main point of mentioning these is because they are one of the things which makes the functional paradigm so powerful: these constructs create a common interface which allows for the different types with which they are associated to

be interacted with in a common way – that is, a lot can be achieved with only a few lines of code (Nikolov, 2016, Ch. 1, Location 4243 & 703). Not just this, but the immutability of the instances within them also makes it easy to perform these operations in parallel and, although this parallelism has not been explicitly implemented in this iteration, the author felt it was worth mentioning them in this section.

4.3 The Presentation Layer

The presentation layer is another one where decoupling and reflection can be observed. The package is accessible via the `PersistenceBridge` class, which uses reflection to instantiate one of the classes which implements the `ConnectionType` trait. This is an implementation of the *Strategy* pattern (Løkke, 2009, p. 80), which allows for the database vendor to be changed without the code having to be recompiled. At the moment, the only dialect implemented is *MySQL*, but a placeholder can already be seen for the *H2* database.

It is also worth noticing that the operations currently being executed by the implementation have a lot of room for improvement: many of the queries and updates which could be grouped together and sent to the database in batch are still being done individually. This does not have a huge impact on performance with a local database, but might cause issues with remote ones. Further iterations of the application should focus on optimising this.

Another pattern which can be observed in this layer is the *Value Object* pattern, which will be discussed in more detail below.

4.3.1 Algebraic Data Types and the Value Object Pattern

Throughout the code, examples of Algebraic Data Types can be seen. These appear in the form of sealed traits and case objects, and are normally used when instances need to be passed around as values, but also contain information which will be relevant to the code. Examples of these can be found in the `ConnectionType` hierarchy, where the number of possible instances for each case class would classify the trait and its subtypes as *Product Types* (Wampler et al., 2015, p. 411). The code listing below illustrates this:

Listing 4: extract of the `ConnectionType` hierarchy showing the case classes used as value objects

```
private[persistence] sealed trait ConnectionType {

  ...

private[persistence] final case class MySql(_dbName: String)
  extends ConnectionType {

  ...
```

```
* implementation of a H2 database
*/

...
```

Algebraic data types could be said to be the natural implementation of the Value Object design pattern. This pattern is used widely for comparison of objects not by their identities, but rather by their values. They consist of small, immutable objects, and the instances of the case classes can be classified as just those (Nikolov, 2016, Ch. 8, Location 3068).

4.4 Application Output

With the current version of the project, the main visible benefit which can be seen is how it summarises income and expenditure by category, and uses this information to display breakdowns by category and simple budgets to the end user. The following is an example of how this can be achieved.

For this example, the user in question wants to upload the following bank statement:

	A	B	C
1	Date	Description	Amount
2	25/07/2017	Ficticious Job July 17	-1542.96
3	25/07/2017	Rainforest Books – Treasure Island	26.54
4	24/07/2017	HEAVEN DIGITAL	18.99
5	24/07/2017	HELP TO BUY ISA	200
6	21/07/2017	DUO AVIAN	557.32
7	17/07/2017	H4G	13.49
8	05/07/2017	Brompton Road Kebab Shop	6
9	06/07/2017	Brompton Road Kebab Shop	6
10	07/07/2017	Brompton Road Kebab Shop	6
11	08/07/2017	Brompton Road Kebab Shop	6
12	09/07/2017	Brompton Road Kebab Shop	6
13	03/07/2017	Honey and Harvey Estate Agents	1000
14	03/07/2017	Doe John STO	-500

Figure 15

As can be seen from the above, there are multiple lines with the same description. This is where the classifier will be most useful.

The user starts the application, and is greeted with the below interface:

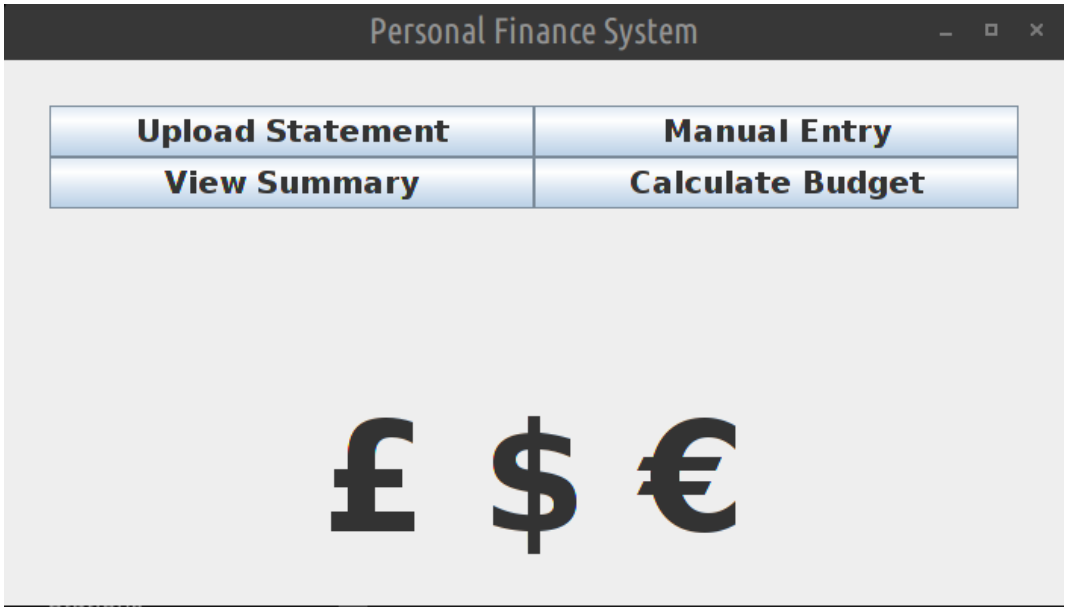

Figure 16

They choose the 'Upload Statement' option, and point to the file containing the statement:

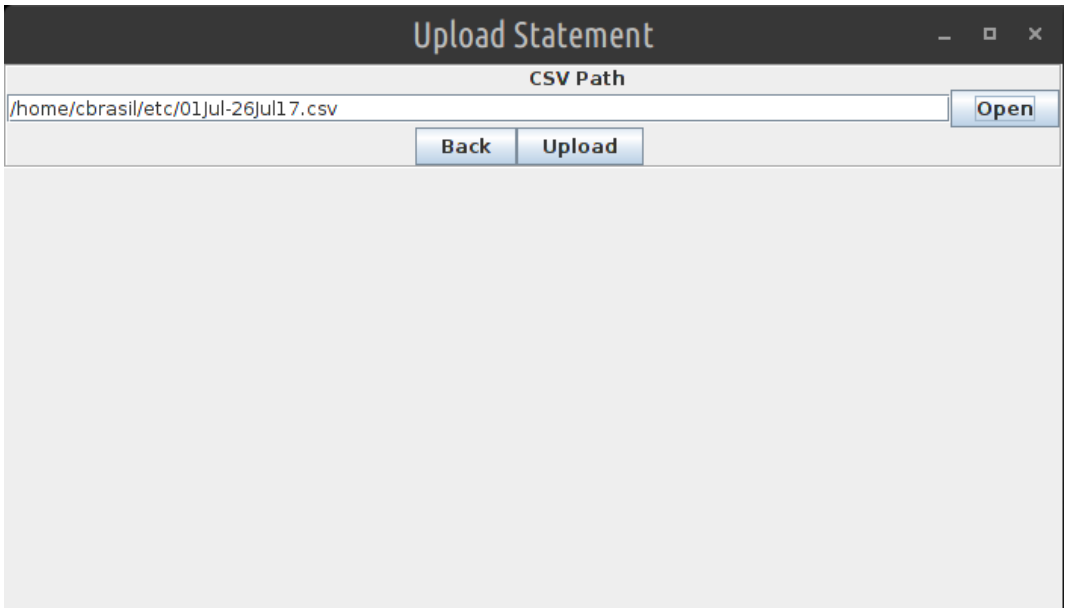

Figure 17

When they click 'Ok', this should trigger the sequence of events which will attempt to classify the transactions. Each time the system encounters a description it cannot match to an existing category, it will prompt the user for input using the `CreateCategory` view. Below is an extract of when trying to classify the income from the user's fictitious job:

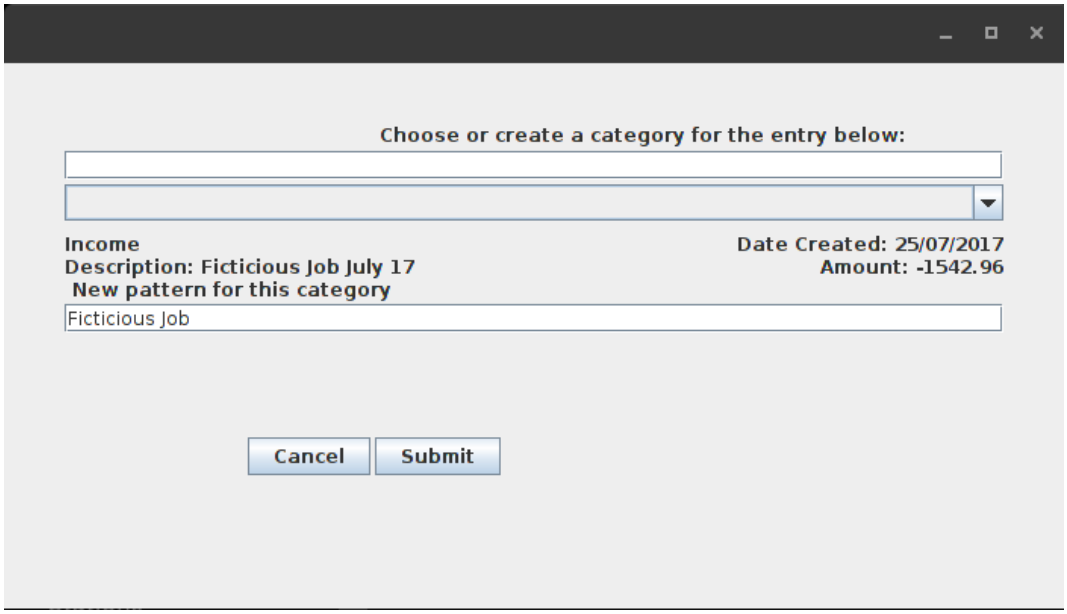

Figure 18

As can be seen from the above, by simply removing the date form the description, the user will ensure that any future income is immediately put in the same category as they choose for this one.

After all the categories have been entered, the user should be able to view a summary by categories for that month, which will look something like the below:

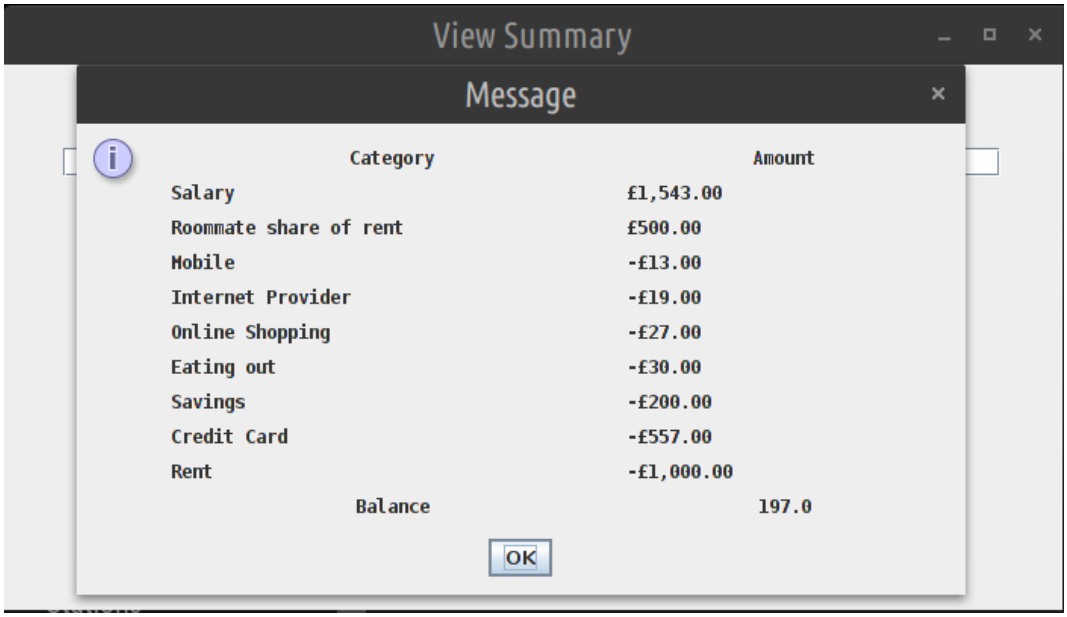

Figure 19

As it can be seen, the balance for that month is positive, showing the user spent less than they earned for that month.

After a whole year's worth of statements have been entered, the user should be able to request a monthly budget. An example of what they would see can be seen below:

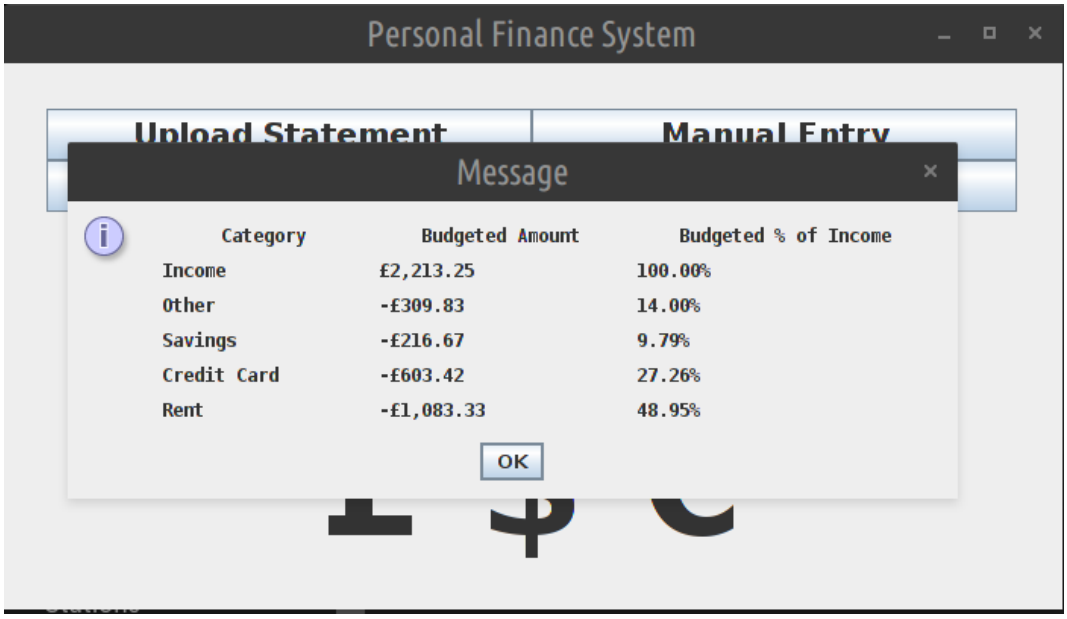

Figure 20

5 Testing

As can be seen in the test suites implemented, the testing for this project was an attempt at Behaviour-Driven Development (BDD), which is a form of Test-Driven Development that focuses on trying to write tests by including descriptions of the expected behaviour, which is also an attempt to make the code more readable (Wynne et al., 2017, Ch. 1). The main library used for testing was `ScalaTest`, "an extensive BDD suite with numerous built-in specs" (Hinojosa, 2013, p. 21). Some of Scala's native features also make it easy to write more readable tests, such as its infix notation, which work well with the members of the `FlatSpec` class and `Matchers` trait.

Most of the automated testing consist of integration tests, since more groups of objects are being tested together but not necessarily the whole application. This was chosen where it felt it would be beneficial to test how well the more heavily integrated objects interacted. Whenever it was appropriate, mocks of other objects were used, as can be seen in the `StringClassifierTester` class. The *Mockito* libraries were used for object mocking, as they work well with `ScalaTest`, with the aid of Scala's `MockitoSugar` trait, to aid with the syntax (Hinojosa, 2013, pp. 102-106).

The automated tests cover mostly the persistence and business logic layers, but unfortunately not much was done with the presentation layer due to time constraints. For the persistence layer, a test database schema was created, and it needs to be loaded into a *MySQL* database before it can be used. The specifications for these need to be entered into `.properties` files before running the tests, otherwise they will fail. Each time they run, any changes made to the test database will be overwritten by the persistence helper, which ensures consistency. The only reason this was done with *MySQL* for the current implementation is that the database is local, therefore performance is not significantly affected. However, for future iterations a portable database would be more suitable for persistence layer testing.

Since TDD was used, in the vast majority of cases the tests were implemented before the classes being tested were written. The classes were then written and refactored until the tests were satisfied. In some cases, further tests were written where it was noticed that more specific behaviours were required of particular classes. After the tests were written, they behaved like a harness which allowed for any further necessary changes to be made with confidence, and helped to speed up the implementation phase (Hinojosa, 2013, pp. x-xi). The result of running the tests at the time of writing can be seen in Appendix IV.

Whenever 'on-the-fly', manual tests were needed, these were done using the Simple Build Tool (SBT) console feature. This feature is a very interesting combination of Scala's Read-Eval-Print-Loop (REPL) with the build tool's ability to quickly integrate all necessary dependencies into a REPL console session (Hinojosa, 2013, p. 1). The advantage of testing in this manner was that it was possible to run a session with compiled code incomplete classes and test different aspects of their behaviour which did not seem fit for automated testing.

6 Conclusions

Although not with as many iterations as originally planned, the main goal of the project has been achieved – that is, to deliver a functional, simple expenditure analyser. The techniques used to develop it should allow for the code to be easily extended and refactored as appropriate, without requiring too much of a rewrite.

The Scala language's multi-paradigm features allowed for most of the variables in the code to be made immutable where appropriate, while still ensuring that the mutable aspects inherited from other languages worked as expected. The functional aspects of the language made it easy for the developer to work well with the immutability, and the flexibility of the data structures also allowed for the focus to be placed on writing good abstractions, while trusting that the language's own implementation of its data structures, especially its collections (Odersky et al., 2016, Ch. 24), would maximise efficiency (as much as a possible).

Regarding the author's perception of *software development best practices*, the advantages of following an iterative cycle were definitely visible, especially those of trying to model the system before beginning the implementation using an appropriate language (in this case, UML). The models in chapter 3 gave very good visibility of possible 'pitfalls', such as too much responsibility being assigned to an entity, or which level of abstraction should be used for each domain aspect being modelled. This allowed for them to be redesigned more easily than would be possible if no planning had taken place.

The benefits of analysis and design patterns were also noticed, and due to these techniques and the SOLID principles (which, although not applied perfectly, seem to have been used up to a reasonable extent), the code can now be extended more easily, and more features can be added with minimum impact being made to existing functionality. This shows, in the author's opinion, the value of these techniques for software development, and why they are considered best practices.

7 Reflections

7.1 Use Case Templates

Originally, no template was used to document the use cases. The intention was to provide better ones at a later iteration, perhaps by researching the ones mentioned by Bennett et al. (2010, p. 157), but unfortunately there was not enough time, so the little there was had to be dedicated to the software itself.

7.2 Nested iterations in Analysis and Design stage

The Analysis and Design stage of the first iteration was delayed due to multiple 'trial and errors' within it. This caused a reflection on whether the development method was truly iterative, or whether or not it was more similar to the Waterfall model.

Still, in the author's opinion, spending more time within the analysis and design stage were very helpful in implementing *SOLID* classes, and as a result decreased the negative impacts that any refactoring in the code base would have caused, were it not to have been done as such. In previous iterations of similar projects by the author, but where the modelling normally done in the analysis/design stages were neglected, classes ended up with too many responsibilities and hard coded dependencies, which made any refactoring very challenging.

7.3 Trade-off between less code duplication and more sub-package independence

In order to keep sub-packages independent, as described in sub-section 4.1, a decision had to be made about increasing code duplication. As can be seen in the current implementation, both `PlayMediator` and `SwingMediator` are supposed to declare the same method signatures, so that there is no conflict when switching between implementations at run time. In order for this to happen, each interface had to be manually typed into its package, and the same will have to happen for any further extensions to the code. An alternative to this would be to allow the sub-packages containing the implementations to see the super-package, and declare the methods once in the `PresentationMediator` interface, but this would have meant sacrificing the concept of sub-package independence already mentioned.

7.4 Layering vs Manual Dependency Injection

One of the original (implicit) goals of this project was to have a hierarchy system, where each sub-package would not depend super package, but super packages could depend on

sub ones. That is, a highly specialised `presentation.swing.frames` package would not depend on elements of the `presentation` package, but the `swing.frames` package would declare interfaces which would then be implemented by `presentation`. What had not been taken into consideration, however, is how the fact that the super package having to implement the interface would make it difficult to truly implement dependency injection: the idea was to have the `InteractionMediator` implement the lower package's interfaces, so that it could be passed to the classes of the specialised package. But this would be a problem when the interface is too specialised. Therefore, a compromise had to be made and the interfaces had to be made more generic.

7.5 Not implemented due to time constraints

The following were not implemented due to time constraints.

7.5.1 Better exception handling and communication with the user

The `PersistenceMediator` class should be handling exceptions which might be thrown by `PersistenceBridge`'s every time the first calls the latter's methods. The intention of allowing the Mediator to catch exceptions was because this could then be passed to the user as informative messages, or be handled internally depending on the nature of the exception (e.g., connection exceptions would have to be handled internally, and exceptions related to user input should be passed to the user). Therefore, some exceptions should be handled by the `PersistenceBridge`, and others by the mediator.

A feature to communicate with the end user was being implemented as well by means of the `presentation.swing.Messenger` object. Unfortunately, there were bugs in the code which did not allow it to work properly, and a solution could not be found in time for the deadline, which meant this feature was not implemented in the MVP.

7.5.2 Validation

A lot of thought has been put into where validation should happen. For example, the constraints of `Transaction`, `Category` and `Entry` which were used to enforce *double-entry* could have been implemented at database or application levels, or both. In the current implementation, however, this constraint is only enforced at business logic layer, and in the `PersistenceBridge.createEntrySet` method of the persistence layer.

This means, unfortunately, that the user is still able to bypass double entry by directly accessing the database, especially since the data is being stored unencrypted. It should be possible to design the application so that the only possible access point to the data will be the application, and that constraints will be enforced at every level, but this will have to be left to a future iteration, which will fall outside of the scope of this report.

Regarding user input validation, the current implementation left a lot of room for improvement in this area. As the current implementation is a desktop app, the risk mainly consists of bad input due to user misunderstandings of the interface, or errors, and not so much of malicious intent. However, it would still be in line with best practices to have an interface that is as "immune" to user-errors as possible. Special characters are not yet being properly handled, but which should be fixed on a future iteration outside of the scope of this report.

Another feature which relates to validation is checking for duplicate entries. As it stands, it is possible to upload the same statement, or make the same manual entry, twice. This is another feature which will have to be implemented at a possible future iteration, outside of the scope of this report.

7.5.3 Implementation of the Strategy Design Pattern for Parser

One of the original intentions of the author was to utilise an implementation of the *Strategy* pattern when loading the user's bank statements into the system. The current implementation uploads data from CSV files, but making use of the Strategy pattern, which allows for different implementations of an algorithm to exist, and for the right one to be chosen while the application is *running* (Nikolov, 2016, Ch. 8, Location 3152), would allow for other formats to be used too. JSON and XML formats come to mind, especially if a version of the application could be made which would allow for it to communicate with a banking system's API – API's (especially RESTful) tend to favour these two formats. This did not happen in this iteration, unfortunately.

7.5.4 The Over-simplicity of the budgeting feature

The current implementation of the budget feature would not work well if the user has overspent: it will simply show an increase the projected income to match the user's expenditure, which would not be a desirable trait in a commercial expenditure analyser – this implies that the application is telling the user to "earn more", rather than spend less. There are better ways to implement this feature, such as decreasing some of the expenditure when budgeting, but the time constraints once again prevented this from being built.

7.5.5 Implementation of Presentation Layer using only Functional Paradigm

After the first iteration of the presentation layer, it was noticed that perhaps it could be fully implemented in a form more close to that of the functional paradigm. That is, make it a point to not use `var`'s, and only `val`'s for members (also not change state using Scala Swing's classes natural mutable fields, such as location, visible, etc – the effects from these could be replicated by copying the values of the instances into new ones). This could have been achieved if the full `presentation.swing` package had been implemented

from the start with this in mind: have an interaction mediator within the package, and then make more use of double dispatch, familiar to the *Visitor Pattern* (Nikolov, 2016, Ch. 8, Location 3943). Then, for the actual flow of the application, a strategy similar to that used by Felleisen et al. (2013, Ch. 5), where each action would trigger a function which changes the state of the application, and then the GUI displaying the new state would be passed recursively to to the main function, ensures that the immutability of the functional paradigm is maintained throughout the presentation layer.

Unfortunately, time constraints were once again too tight for this to be fully implemented.

7.5.6 Decimal Precision

The application was implemented using `Double` as the data type. Unfortunately, after testing it with a larger dataset, it was noticed that this was affecting the precision of the data. Due to time constraints it was not possible to switch all the necessary classes to `BigDecimal`, or another similar implementation which would allow for decimal precision to be achieved. This should happen in a future iteration, however.

7.5.7 Aesthetics

For the current implementation, the aesthetics of the front end were heavily neglected. During development of this version of the product, although some attention was being given to achieve symmetry – one of the aspects of classical aesthetics – this was only possible in the limited time because *Scala Swing* makes symmetry almost intrinsic to the organisation of its `BoxPanel` class. All other aspects of classical and expressive aesthetics (Lavie and Tractinsky, 2004, cited Benyon et al., 2005, p. 102) had to be left for a future iteration.

References

Bauer, Christian, King, Gavin, and Gregory, Gary (2016). *Java Persistence with Hibernate*. Second Edition. Manning Publications Co.

Bennett, Simon, McRobb, Steve, and Farmer, Ray (2010). *Object-oriented systems analysis and design using UML*. Fourth Edition. McGraw Hill Higher Education. ISBN: 978-0-07-712536-3.

Benyon, David, Turner, Phil, and Turner, Susan (2005). *Designing interactive systems: People, activities, contexts, technologies*. Pearson Education.

Boczko, Tony (2012). *Introduction to Accounting Information Systems*. Pearson Custom Publishing.

Dawson, Christian W (2009). *Projects in computing and information systems: a student's guide*. Second Edition. Pearson Education.

Evans, Eric (2004). *Domain-driven design: tackling complexity in the heart of software*. Addison-Wesley Professional.

Felleisen, Matthias, Van Horn, David, Barski, Conrad, et al. (2013). *Realm of Racket: Learn to Program, One Game at a Time!* No Starch Press.

Fowler, Martin (1997). *Analysis patterns: reusable object models (OBT)*. Kindle Edition. Addison-Wesley Professional.

— (2004). "Inversion of control containers and the dependency injection pattern". In:

Gamma, Erich, Helm, Richard, Johnson, Ralph, and Vlissides, John (1995). *Design patterns: elements of reusable object-oriented software*. Addison-Wesley.

Hinojosa, Daniel (2013). *Testing in Scala*. "O'Reilly Media, Inc."

Holmes, Simon (2016). *Getting MEAN with Mongo, Express, Angular and Node*. Manning Publications Co.

— (2018). *Getting MEAN with Mongo, Express, Angular and Node*. Second Edition, Early Access, V7. Manning Publications Co.

Løkke, Fredrik Skeel (2009). "Scala and Design Patterns-exploring Language Expressivity". In: *Aarhus Universitet, Datalogisk Institut* 44.

Maier, Ingo (2009). "The scala.swing package". In: *Scala Improvement Process (SID)* 8.

Martin, Robert C (1996). "The dependency inversion principle". In: *C++ Report* 8.6, pp. 61–66.

Nikolov, Ivan (2016). *Scala Design Patterns*. Kindle Edition. Packt Publishing Ltd.

Odersky, Martin, Spoon, Lex, and Venners, Bill (2016). *Programming in Scala*. Third Edition, Kindle eBook. Artima Press.

Payments UK Management Ltd. (2017). *UK Payments Markets Summary*. URL: https://www.paymentsuk.org.uk/files/puk-uk-payment-markets-2017-summarypdf-0 (visited on: 18 Dec. 2017).

Quicken Inc. (2017). *About Us*. URL: https://www.quicken.com/about-us (visited on: 17 Dec. 2017).

Wampler, Dean and Payne, Alex (2015). *Programming Scala*. Second Edition. "O'Reilly Media, Inc."

Wood, Frank and Robinson, Sheila I (2004). *Book-keeping and Accounts*. 6th Edition. Pearson Education.

Wynne, Matt, Hellesoy, Aslak, and Tooke, Steve (2017). *The cucumber book: behaviour-driven development for testers and developers*. Pragmatic Bookshelf.