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

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

May 2018

1 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

2 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.

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 listed in Chapter **??**: Chapter 3 outlines the identified requirements which were used as motivation for the system to be developed; the contents of Chapter 4 shows further analysis and concomitant design of the system and the solutions it brings; Chapter 5 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).

For this project, 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 1, 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.

2.1 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 was 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.

2.2 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.

2.2.1 Analysis and Design Patterns

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". This project will therefore attempt to utilise patterns where appropriate in order to prove 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.

The concept of *domain* here is being used, as defined by Evans (2004, p. 2), as the "activity or interest of its user" – the "subject area to which the user applies the program".

Analysis patterns will be used "when trying to understand the problem" domain (Fowler, 1997, Section 1.1). 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).

3 Requirements

3.1 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.

3.2 Functional Requirements

Based on the description above, a few functional requirements were identified. They are represented in the diagram on section 3.2.1 and the list, wireframes and activity diagrams on section 3.2.2.

3.2.1 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:

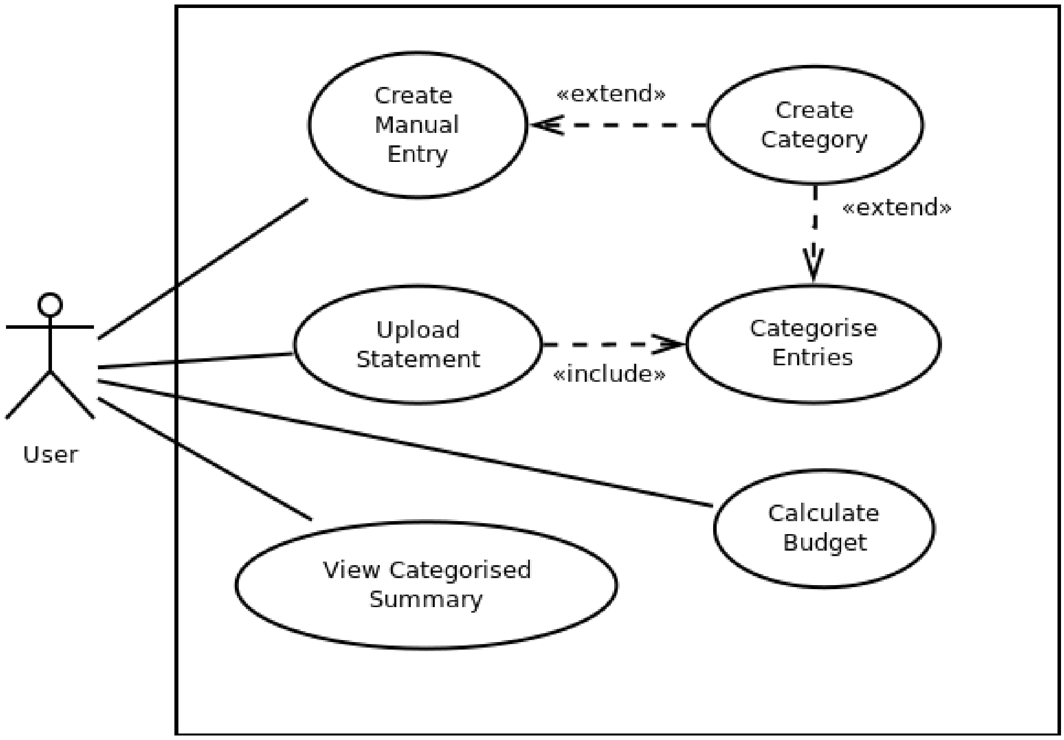

Figure 1: Use Case Diagram

3.2.2 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 IV.

3.2.3 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:

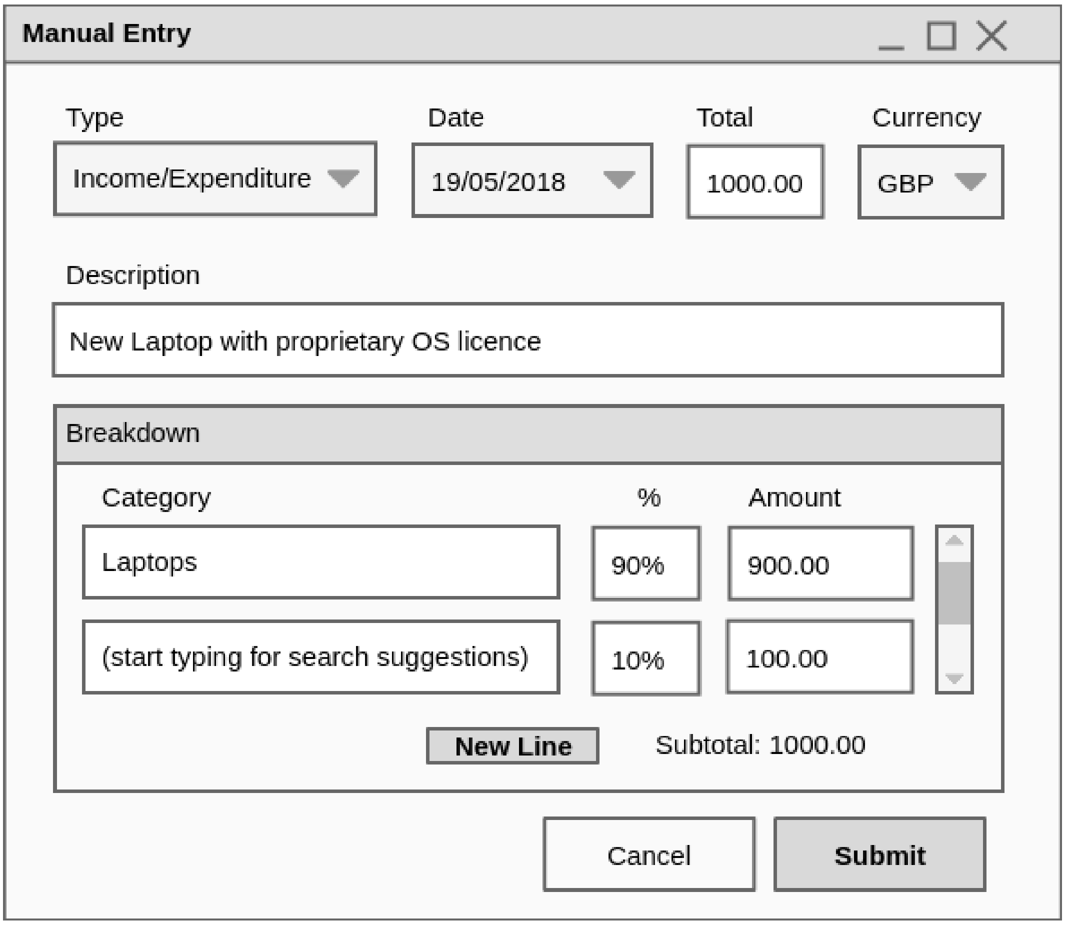

Figure 2: User interface wireframe for *Create Manual Entry* use case

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

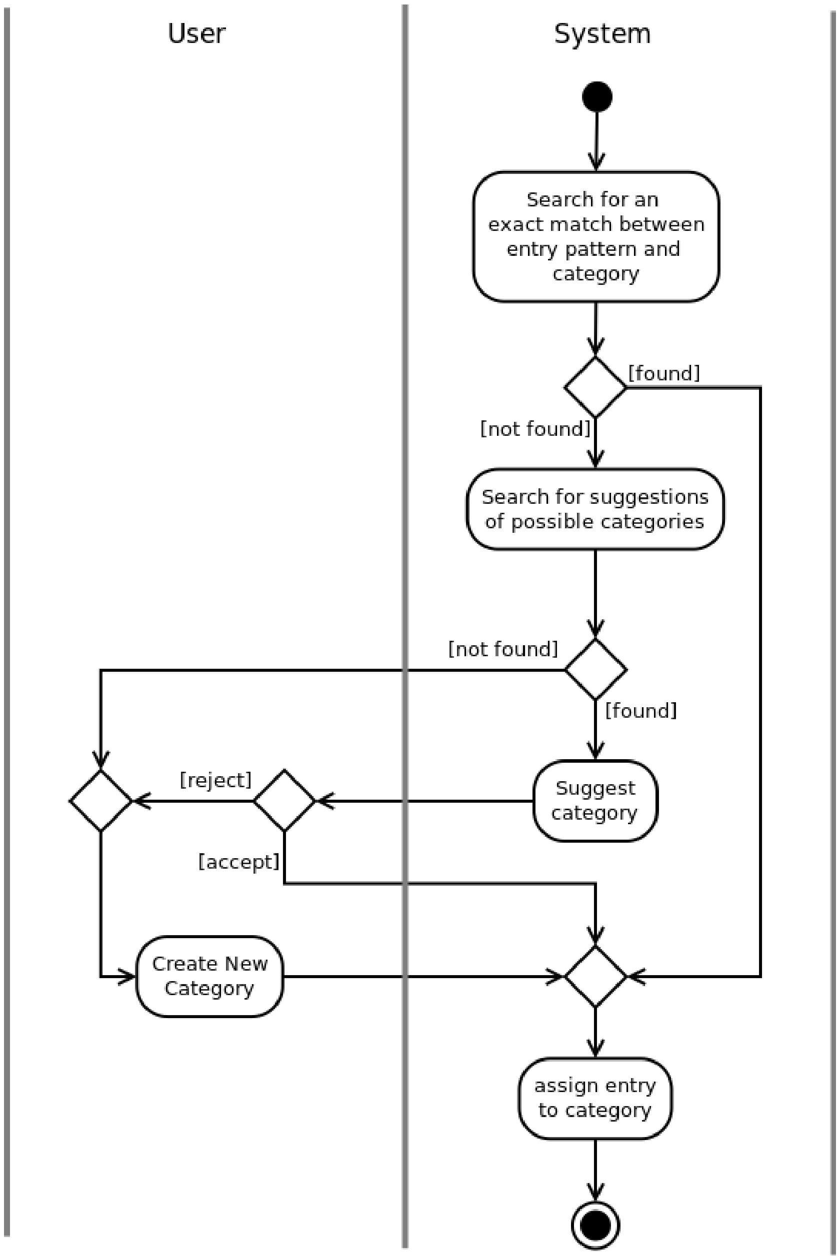

Figure 3

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:

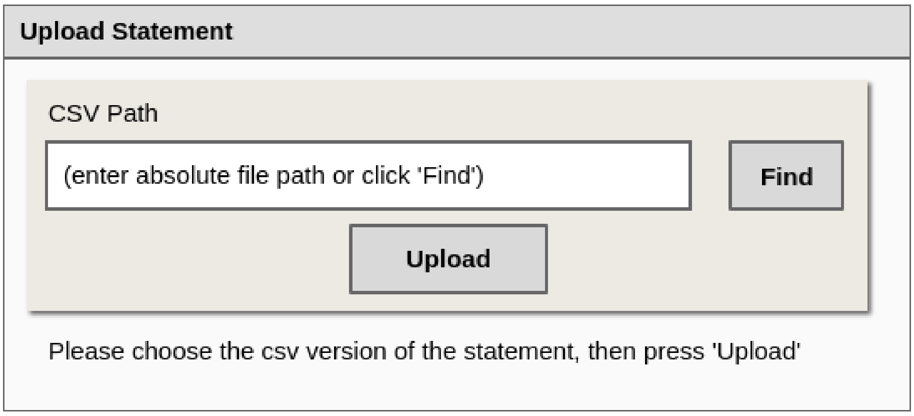

Figure 4

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:

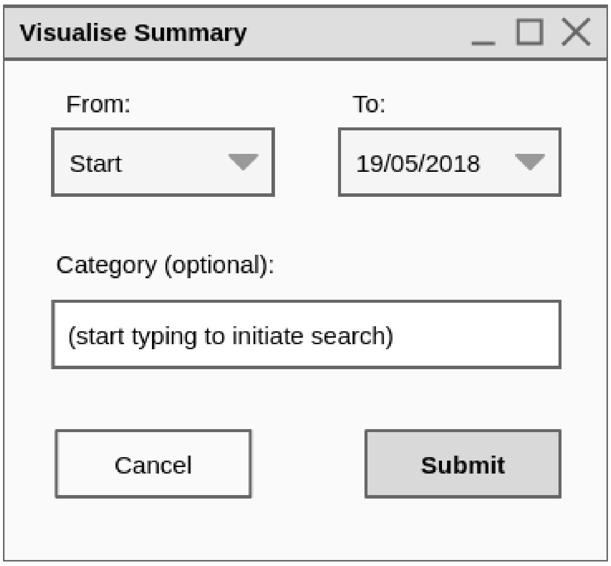

Figure 5

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.

3.3 Non-Functional Requirements

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

4 Analysis and Design

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 similar also to what has been described by Holmes (2016, p. 3) as full stack development for an application using a Model-View-Controller (MVC) architectural pattern.

Most of the aesthetic designs of the presentation layer happened in chapter 3.2, so this one will focus on the analysis and design of the business logic and persistence ones.

4.1 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):

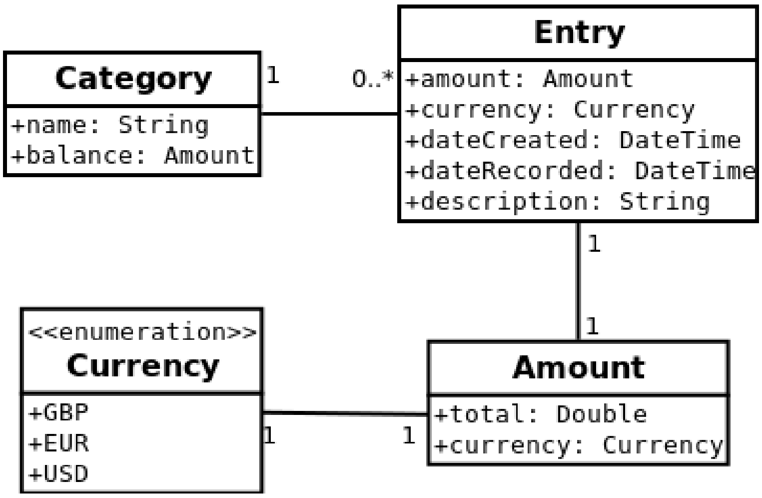

Figure 6

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):

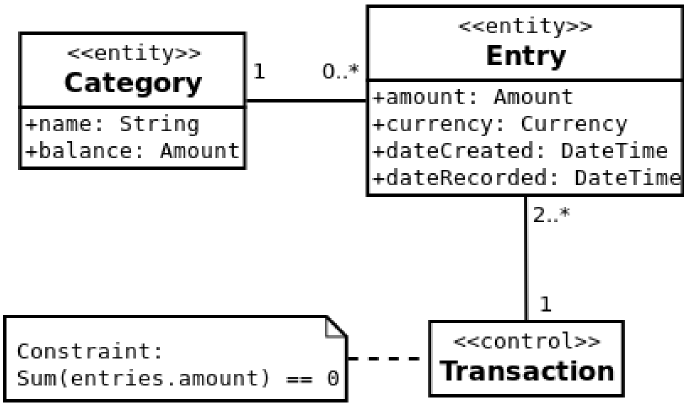

Figure 7

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 3. 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.

4.1.1 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):

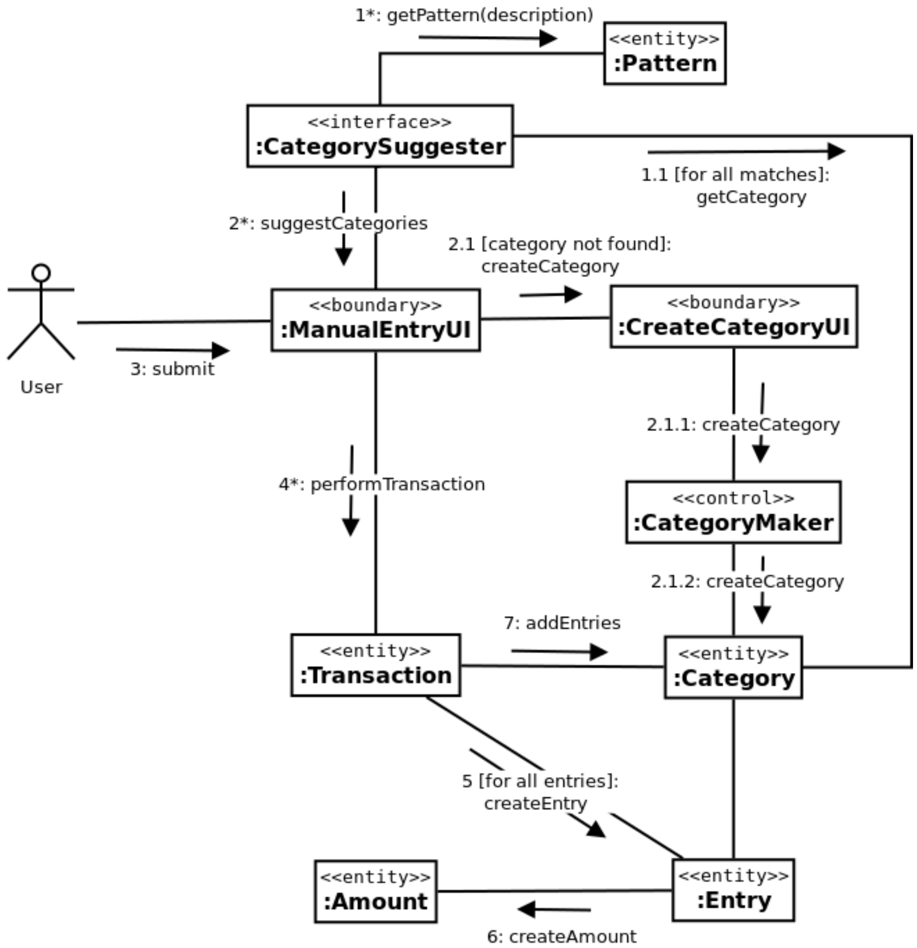

Figure 8: Communication Diagram for the *Create Manual Entry* use case

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.

4.1.2 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:

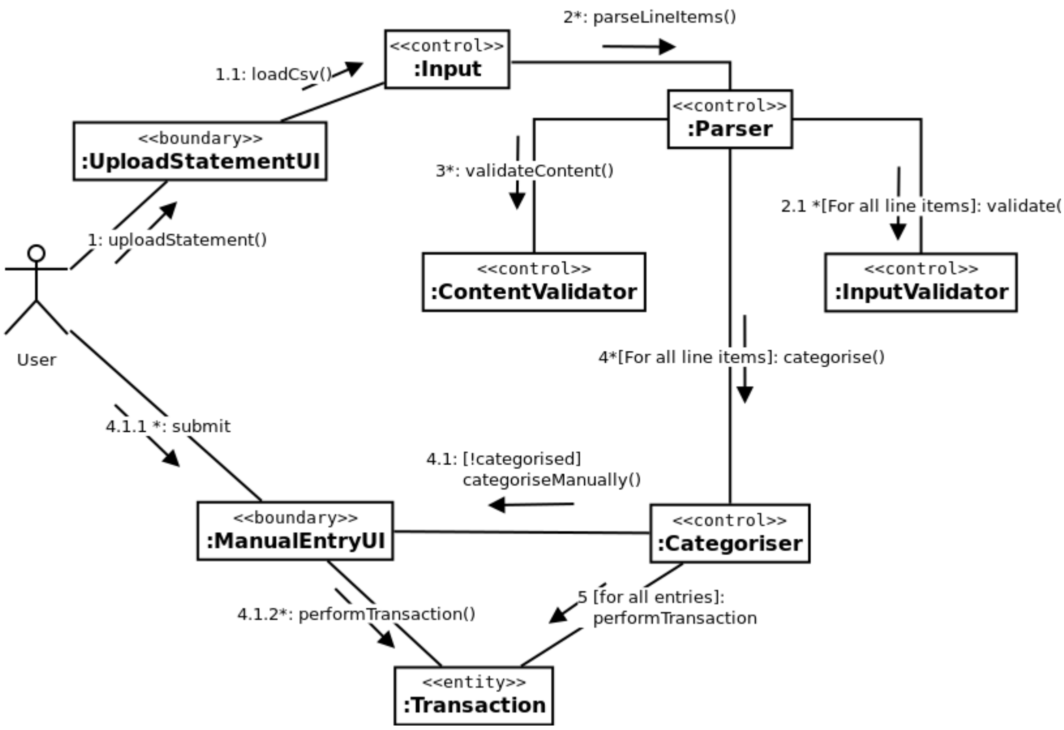

Figure 9: Communication Diagram for the *Upload Statement* use case

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

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

4.1.3 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.

TODO: see Bennett et al., 2010, p. 262: sequence diagrams are for simple interactions. Move this to one of those

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

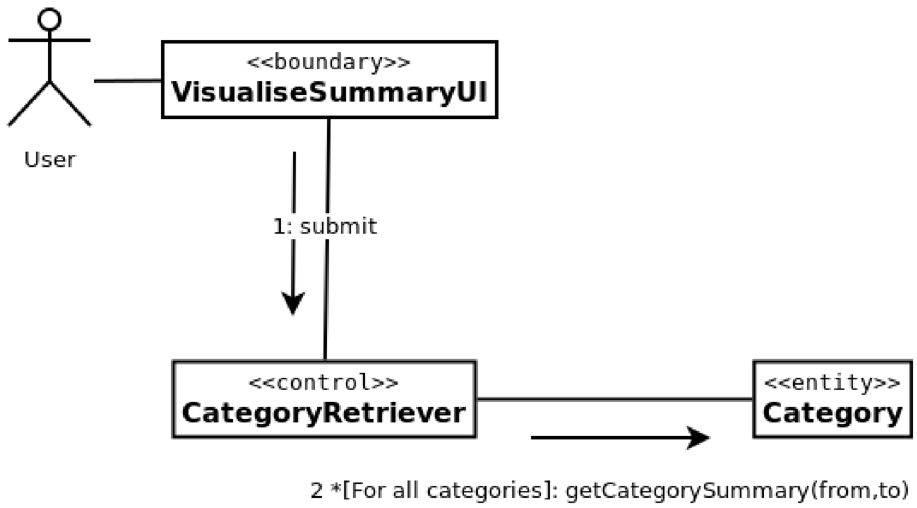

Figure 10: communication diagram for *Visualise Categorised Summary* use case

4.1.4 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.

TODO: see Bennett et al., 2010, p. 262: sequence diagrams are for simple interactions. Consider moving this to one of those if the transaction can be simplified

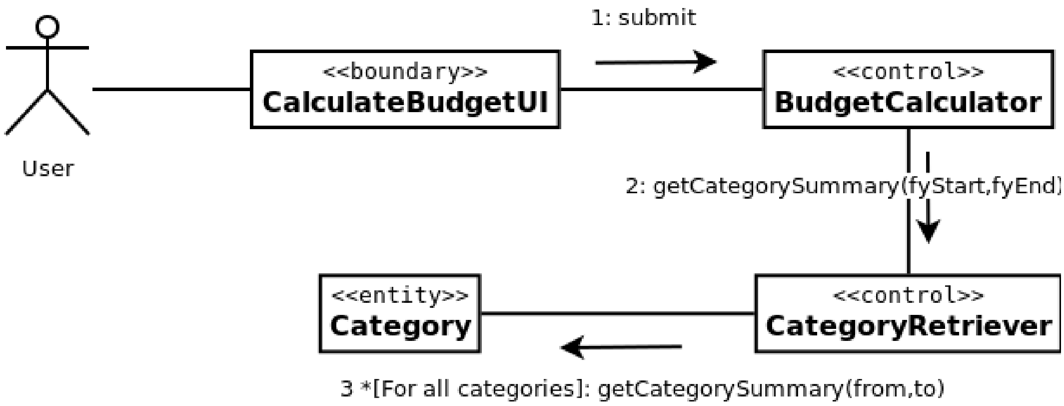

Figure 11: communication diagram for *Calculate Budget*

4.1.5 Final Class Diagram

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

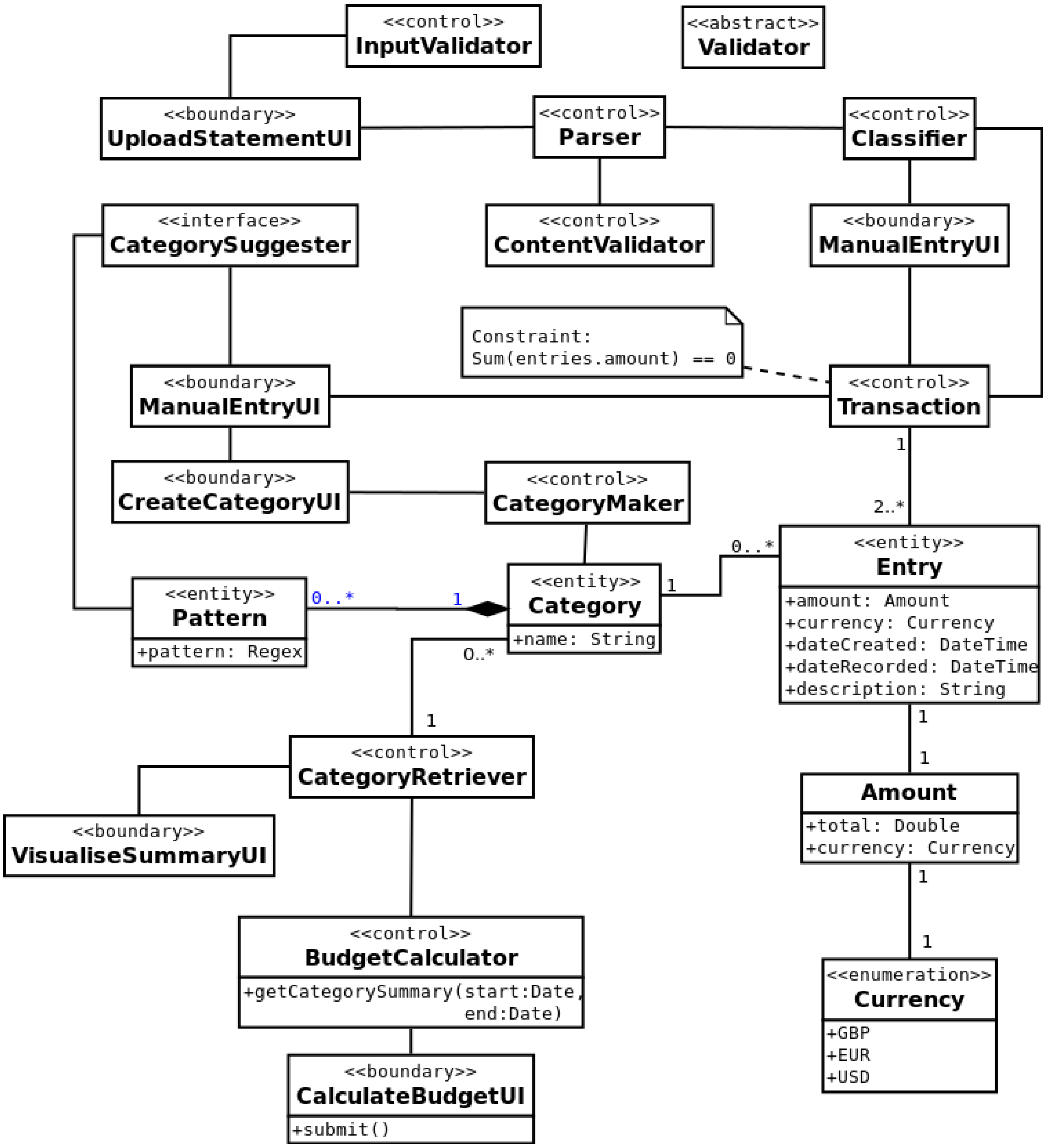

Figure 12: Class diagram of the most relevant classes of the Business Logic Layer

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.

4.2 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

4.3 Database Diagram

TODO: redo this diagram – include the currency tables The UML diagram below shows the design of the database schema for the application:

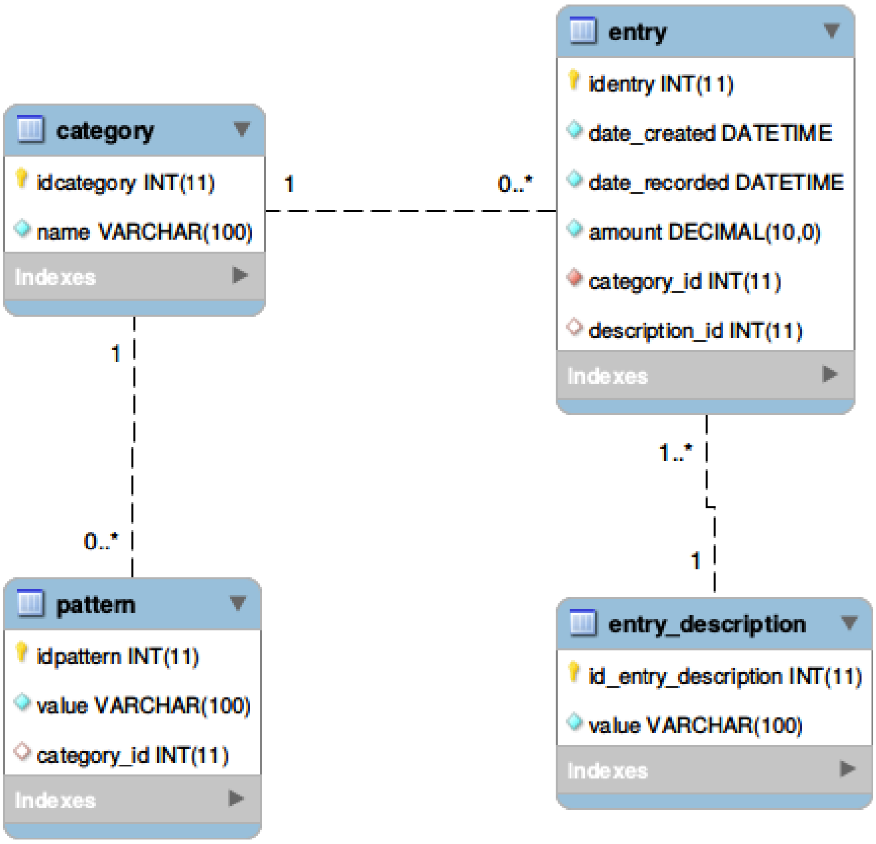

Figure 13: UML Diagram of database schema

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 usage.

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.

5 Implementation

The initial MVP was designed with slight differences from the design outlined in chapter 4. The most visible one is the fact that what is mainly the Mediator pattern (Nikolov, 2016, Ch. 9, Location 3594) was used in order to implement the dialogue between the presentation and other layers using a feature similar to the MVC architecture. 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. However, a mechanism was put in place which could allow for this to happen.

In the implementation used for this project, the components were segregated into packages 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- (or outer) packages, then, implement the interfaces of the sub-packages, which allows them to interact with the sub-package components. The diagram below (Figure 14) illustrates this using the interaction between the `InteractionMediator`, `SwingAmbassador` and `swing` sub-package of the `presentation` package:

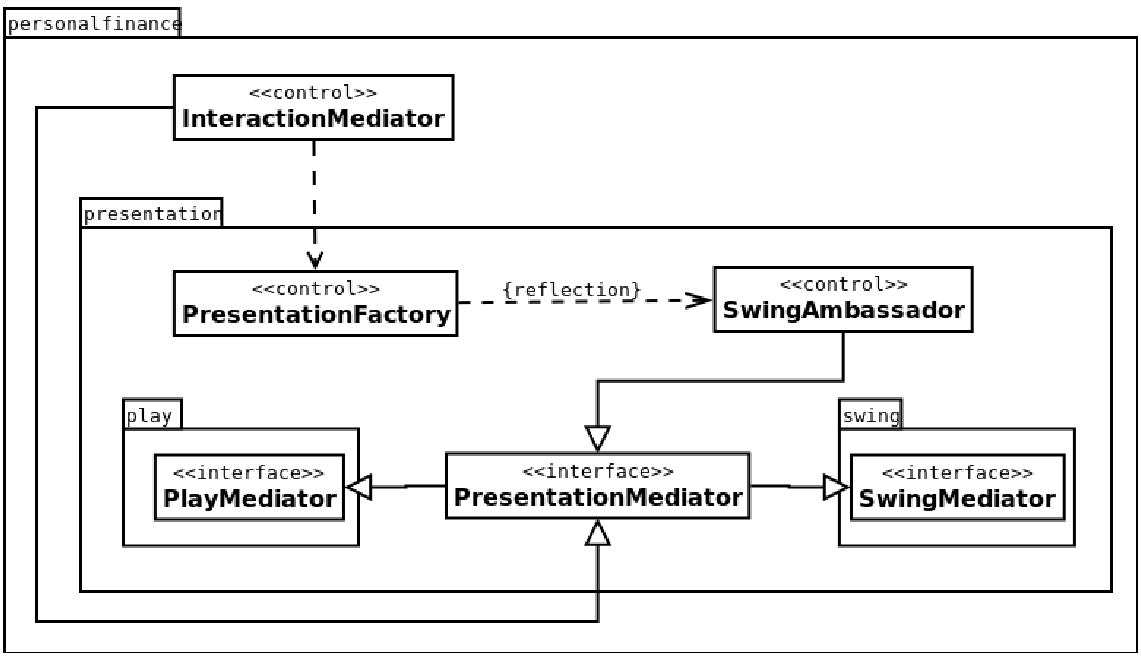

Figure 14

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 code, 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 implementation of the view it is interacting with. The *Factory Method* pattern implemented on PresentationFactory, which uses reflection to determine what implementation to deliver, allows for the view itself to be decided with the use of an external .properties file. This allows for the actual dependency 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.

5.1 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 the class 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 feature.

One of its many utilisations in this project 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 1: extract of the Transaction class showing the copy() method in action

```
    transactionUnits.map(addEntries)
  }

  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)
```

5.2 Presentation Layer

In the presentation layer, for the first iteration, the *Scala Swing* package was used for building most its aspects. The package 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 application starts by implementing the `MainMenu` class, which extends the `SimpleSwingApplication` abstract class from the `scala.swing` package. The class has an implementation of a *main method*, therefore it acts as an entry point for the application to run. It also contains a `MainFrame`, which is a Swing `Frame` – "a window containing arbitrary data" (Odersky et al., 2016, Ch. 34, Section 34.1)– which switches off the application when closed. It also has implementations of other methods to allow the application to close gracefully (Maier, 2009, p. 2 & 3), so with these aspects implemented, it allows development to be focused on functionality more relevant to the application's logic itself.

5.3 Business Logic

TODO: syntax highlighting for code listings

TODO: write about how using properties file is a good way to decide otherwise static behaviour at runtime – is this dependency injection?

TODO: Write about how one of the advantages of scala is how it tries to enforce the Universal Access. For example, Arrays and Lists implement the interface (which I think is Iterable) which allows the user to completely ignore how the data is implemented and access both as lists

TODO: write about why Validator has an auxiliary method for type – generics would affect the interface. The Validator was created as a trait in the first place to allow for lesser coupling between the classes which need validation

TODO: write about why I chose to do integration tests rather than unit

TODO: Read the lightbend guide https://www.lightbend.com/lagom-framework to Scala and Microservices

TODO: see if the below still makes sense The constraints from `Transaction` and `Pattern` were implemented in the business logic code as follows:

Listing 2: method to validate `Transaction` entries

```
  new TransactionEntriesValidator,
  new TransactionUnitValidator
)) {

  /**
    * This method validates and executes a transaction executed by a user
    * Decided to use dependency injection on method here, so that one Transaction
    * object can still be reused with multiple Category/Entries pairs
    */
  def execute(transactionUnits: Seq[TransactionUnit]): Seq[Category] = {
```

Listing 3: snippet of `Pattern` showing requirement for it to have at least one Category

```
/**
  * This is the pattern class which will be a part of Category
  */
case class Patterns(list: List[Pattern])

case class Pattern(value: String, id: Option[Int] = None)
```

As Fowler (1997), one of the classes should be responsible for keeping track of the relationship. As seen above, it was decided that each `Category` will keep track of its patterns, and this will at the same time enforce the constraint at the application layer level.

Classifier was implemented with longest matching prefix: 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 girlfriend" 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.

TODO: talk about the decision to create constructor dependencies to traits rather than classes directly, as is the case with `DateStringParser`

TODO: talk about choosing not to user the Builder pattern for the Entry class as it would turn into boilerplate town

TODO: talk about the small amount of input validation due to the first version being for a desktop app only, but that if a hosted version were to be made available more validation would be added

TODO: talk about the decision to have single methods to commit to the database one entry at a time, which was because the database is local and because it gives more control over when something goes wrong, both for debugging or informing the end user. In future iterations, though, the plan was to use batch versions as exemplified here https://docs.oracle.com/javase/tutorial/jdbc/basics/p

Explain decision to make dependencies only flow in one direction – namely, towards the more specialised packages (aside from the dependencies on precompiled libraries, such as java.util and the ones listed on sbt) – so that the specialised modules could be extracted and reused in a more modular fashion. Explain also why the modules in the default package are implementing all similar interfaces in the more specialised ones, so that changes in the more specialised ones which could affect dependencies will also have to be implemented at the higher level, thus causing the flow to be maintained. E.g., each package declares PropertiesLoader, and the default package.

But: this has not been tested yet. Some classes such as PropertiesLoader

are private to the personalfiance package. This means that external pacakges cannot import it directly, but would it be indirectly imported as a dependency when importing other classes? Also, does it matter, since the dependency is actually in an interface which is later implemented by the personalfinance package?

Talk how the above is a risky move, since Scala's mixin composition has this habit of picking up the implementation of the last trait mixed into it, so in this case you get PresentationMediator inheriting form SwingMediator and then PlayMediator – although both are traits, this could cause problems in the future if the Play one implements something which Swing doesn't, so basically these traits must always be the exact same

talk about the reason why the presentation mediator's methods returning Unit being because this way they can stay as implementation-agnostic regarding the other layers as they possibly can, again thinking that the interface needs to be able to change without the code being recompiled

5.4 The Bridge Pattern

TODO: review this. It all changed after the 20th of April The Bridge pattern allows for an abstraction and its implementation to be decoupled, so that both can vary independently (Nikolov, 2016, Ch. 7,Location 2699). In this project, the bridge pattern can be seen implemented in the *presentation* and *persistence* layers. In the first, the `Presentation` class is linked to the `PresentationBridge` trait by aggregation. The `Swing` class implements this trait and refers to the *swing* package for the implementation. This was done so that the implementation of the *presentation* layer can change easily at runtime – for example, if there are a hosted and a portable version of the application, the first which runs on a web framework and the latter with a desktop package such as Swing, the same application can be implemented to both without the code having to be recompiled.

Another feature which facilitates this is the fact that the clients – the `Presentation` class – uses a combination of a `java.util.Properties` object and *reflection* to determine which implementation it will run. This will also aid the above goal of having the behaviour of the application change without it having to be recompiled.

5.5 The Strategy Pattern

Thanks to Scala's mixin traits, the Strategy Pattern was implemented successfully in the `PresentationMediator` class with only a few lines of code. What happens here is that the combination of the Strategy and Factory patterns are allow enough flexibility that the more specialised packages do not have to depend on the packages which envelop it – within the `presentation` package there is a `swing` package and a placeholder `play` package for the front-end implementations. So long as the outer packages implement the interfaces for each specialised package,

5.6 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

```
package personalfinance
package persistence
package connections

/**
  * This trait needs to be implemented by classes which will hold the hardcoded
  * references to the query dialect they represent
  */
private[persistence] sealed trait ConnectionType {
  protected val category = s"$dbName.category"
  protected val pattern = s"$dbName.pattern"
  protected val entry = s"$dbName.entry"
  protected val entry_description = s"$dbName.entry_description"

  def dbName: String

  def queryForAllCategoriesAndPatterns: String

  def queryForACategory(name: String): String

  def queryForCategoryPatterns(id: Int): String

  def createCategoryOnly(name: String): String

  def createPatternOnly(categoryId: Int, patternValue: String): String

  def createEntryDescriptionPS: String

  def getEntryDescription(description: String): String

  def getSummaryPS(): String

  def createEntryPS: String
}
```

```
/**
  * this is an implementation of MySql's dialect
  */
private[persistence] final case class MySql(_dbName: String) extends ConnectionTyp

  override def dbName: String = _dbName

  override def queryForAllCategoriesAndPatterns: String =
    s"""select *
      |from $category cat
      |   left join $pattern pat
      |     on cat.idcategory = pat.category_id;""".stripMargin

  override def queryForACategory(name: String): String =
    s"""select idcategory,name
       |from $category cat
       |where cat.name = '$name';
     """.stripMargin

  override def queryForCategoryPatterns(catId: Int): String =
    s"select idpattern,value,category_id from $pattern where category_id = $catId"

  override def createCategoryOnly(catName: String): String =
    s"insert into $category (`name`) values ('$catName')"

  override def createPatternOnly(categoryId: Int, patternValue: String): String =
    s"insert into $pattern (`value`,`category_id`) " +
      s"values ('$patternValue', '$categoryId');"

  override def createEntryDescriptionPS: String =
    s"insert into $entry_description (`date_created`, `date_recorded`, `value`) va

  override def getEntryDescription(description: String): String =
    s"select id_entry_description,value from $entry_description " +
      s"where value = '$description'"

  override def createEntryPS: String =
    s"insert into $entry (`amount`, `category_id`, `description_id`, `currency_id`
      "values (?, ?, ?, ?)"

  override def getSummaryPS(): String =
    "select cat.name as Category,sum(amount)*-1 as Total " +
    "from category cat left join entry on entry.category_id = cat.idcategory " +
    "join entry_description on entry.description_id = entry_description.id_entry_d
    "where date_created >= (?) and date_created <= (?) " +
    "group by cat.name " +
    "order by sum(amount)"
}

/**
  * This is a placeholder for a possible future
```

```
  * implementation of a H2 database
  */
private[persistence] final case class H2(_dbName: String) extends ConnectionType {

  override def dbName: String = _dbName

  override def queryForAllCategoriesAndPatterns: String = ???

  override def queryForACategory(name: String): String = ???

  override def queryForCategoryPatterns(id: Int): String = ???

  override def createCategoryOnly(name: String): String = ???

  override def createPatternOnly(categoryId: Int, patternValue: String): String =

  override def createEntryDescriptionPS: String = ???

  override def getEntryDescription(description: String): String = ???

  override def createEntryPS: String = ???

  override def getSummaryPS(): String = ???
}
```

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).

6 Testing

TODO: write about Scala's own implementation of Mockito as one of the language features

TODO: write about why I chose BDD as my form of TDD, if there's space write about the BehaviourTester class. Include Scala's infix notation as one of the reasons

TODO: write about how there is very limited testing in the presentation layer due to the fact that I do not have much experience with testing front end

TODO: talk about how testing for the persistence layer was done with MySql because the present version uses a local db, but that in the future for performance reasons they should be done using a self-generated portable db version instead (such as H2)

TODO: had to make a decision when testing the persistence regarding the reloading of the whole DB from the dump after performing Create/Update/Delete only. Reloading it every time using 'before' was just taking too long, so to improve performance I redid it in an unelegant way. The alternative of having one test suite for reading and another one for writing was not working also, as many problems were happening, possibly due to so many classes overwriting the database concurrently

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. Appendices **??** and **??** show a examples of different models which had been incorporated into the final design, but which were later changed even before the implementation started – that is, even before any coding was done. This caused a reflection on whether the development method was truly iterative, and 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 Dependency Injection

Very often during the implementation it was felt that better dependency injection could be achieved. In classes such as those found in the `TransactionsValidators.scala` file there was constant hard-coded dependency to the `validation` package. In this instance, a framework such as *Spring* or *Guice* would have been useful, but unfortunately time constraints made it unlikely for the author to implement these into the project. As a result, there is more tight coupling than there could be, but wherever possible an effort has been made to pass the dependencies as constructor or method parameters, so as to facilitate testing, among other things.

7.4 Validation

TODO: see if this is still the case at the end of the project 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. Initially they were implemented in the business logic.

7.5 Design Patterns

TODO: see if this is still the case at the end of the project Instead of the original plan to actually apply design patterns by writing implementations of those manually, the author noticed that the final version of this project currently uses mainly the patterns already available in the *Scala* language.

7.6 Implementation of the Strategy Design Pattern for Parser

TODO: see if this is still the case at the end of the project One of the original intentions of the author was to utilise an implementation of the Strategy Design 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.

Unfortunately, the time constraints prevented these to be successfully implemented and tested at this time.

7.7 Implementation of Presentation Layer using only Functional Paradigm

THIS MAY HAVE BEEN CHANGED. LAST COMMIT WHERE TRUE WAS b0bae376264f 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.

7.8 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.

7.9 Not Implemented

Unfortunately, the only feature actually delivered was a very crude implementation of the manual entry feature. The business logic for classification, which would enable the uploading of statements, has also been created, but the linking of it to the GUI was not done in time.

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. Unfortunately, time constraints prevented this from happening.

TODO: talk about having to make a decision between making the specialised modules more independent by having them depend on code in the same layer, and having code duplication, such as with the Mediators which depend on two different traits

talk about the decision to sometimes use pattern matching with objects and at other times use enum classes (as with the objects which extend KitName in the presentation layer vs the EntryType in the root personalfinance package), and that although here they have been used interchangeably, a level

of consistence would have been improved readability in a different situation

Regarding the fact that the same transaction can be repeated multiple times with manual entry, this was allowed to happen because it is not clear whether it is impossible for multiple transactions in a statement to have the exact same creation date, description and value. The same is true for manual entries: if the user wants, they can enter as many entries with the same amount as they want

The lack of experience of the author really showed when it came to getting individual categories from the user. The idea of making subpackages unaware of super packages led to "awkward" uses of tuples and maps as parameters on the user interface, and a somewhat unelegant way of handling the remaining uncategorised entries after a new category has been created: since every method has a Unit value, the whole block of entries had to be passed around as messages so that the state would not be lost. Hence why there are so many Seq String String String String flying around.

Perhaps due to what is a combination of inexperience and lack of more detailed preparation regarding method signatures, there is a bit of (for lack of a better term) 'awkwardness' around method signatures, return types and constructors. For instance, Transaction.execute takes Seq TransactionUnit, but at times it is clear that only two transaction units are going to be entered, but it is still necessary to make them into a sequel. Or when Classifier makes a transaction unit, it still needs to wrap the category and entry into a Seq before it can create the method. Although it all works, there could be better ways to engineer it so that it is clear from the contract and the fields in the class what is happening inside of it

The way the matching of an entry description against a pattern is being done should be by: first sort all the entries by longest description; then categorise, create patterns etc; then when actually classifying similar entries in the next month, start by the longest transactions, and match them first against the longest patterns, then the shortest. This way more generic patterns such as 'laptop' would not match first with an entry such as 'laptop for girlfriend'

Although originally there did not seem to be any advantages on implementing the 'Amount' field in the 'Category' class, as suggested by Fowler (1997) when referring to the 'Account' analysis pattern, it became apparent that it would have been beneficial to have it for the budgeting feature. If each category could have their own amount, then instead of passing around Tuples of String and Double, instances of 'Category' could have been passed instead, each of which containing their own subtotal (as per Appendix III) during the period in question.

The current implementation of the budget feature would not work well if the user has overspent: it will increase the projected income to match the user's expenditure, which would not be a desirable trait in a commercial

expenditure analyser. 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.

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.

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.

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

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.

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

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.