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 business, it could be argued that the same concept could be applied for 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. There must also be a feature to allow the user to view summaries of the income and expenditure over a period of time, as well as one to forecast budgets for future periods based on the "financial behaviour" analysed.

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 7: 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.

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).

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 categorised 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. This 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 'debit' £50 to the *Bank* category, and then 'credit' £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.

A feature to estimate tax would also be useful. Initially, this can be based on the system used to calculate personal tax in the UK, where all of an individual's income is added up, and depending on which threshold it reaches, tax is deducted at that level. For example, for an individual earning £75,000 per annum from their job where he or she is a company employee, initially their tax free allowance would be deducted from the gross figure, then the basic rate is deducted from it up to its limit, then higher rate would be deducted from the rest. However, if they have any income from sources which have special taxation rules, such as interest on savings, these would be deducted right after the personal allowance – that is, income from special taxation sources get added to the gross income figure, but tax from them is deducted at a different rate and before the base tax. (**TODO: CITATION NEEDED**).

There should be an option to allow a user to determine whether a transaction is taxable. For income, the user should also be able to determine if it is already net of tax (taxed at source) or if it's the gross amount, since this would influence the tax calculation. Also, on expenditure, certain costs such as memberships to professional organisations are deductible, so the user should be able to indicate this when they are registering the category.

Furthermore, for the tax feature, it is important to emphasise that not all of an individual's income will get taxed under income tax. For example, profits on sales of shares gets taxed as capital gains tax. Since this type of tax is outside of the scope of this application but would still appear on a user's list of transactions, the system needs a way to highlight these so as for them not to be included in the tax calculation.

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:

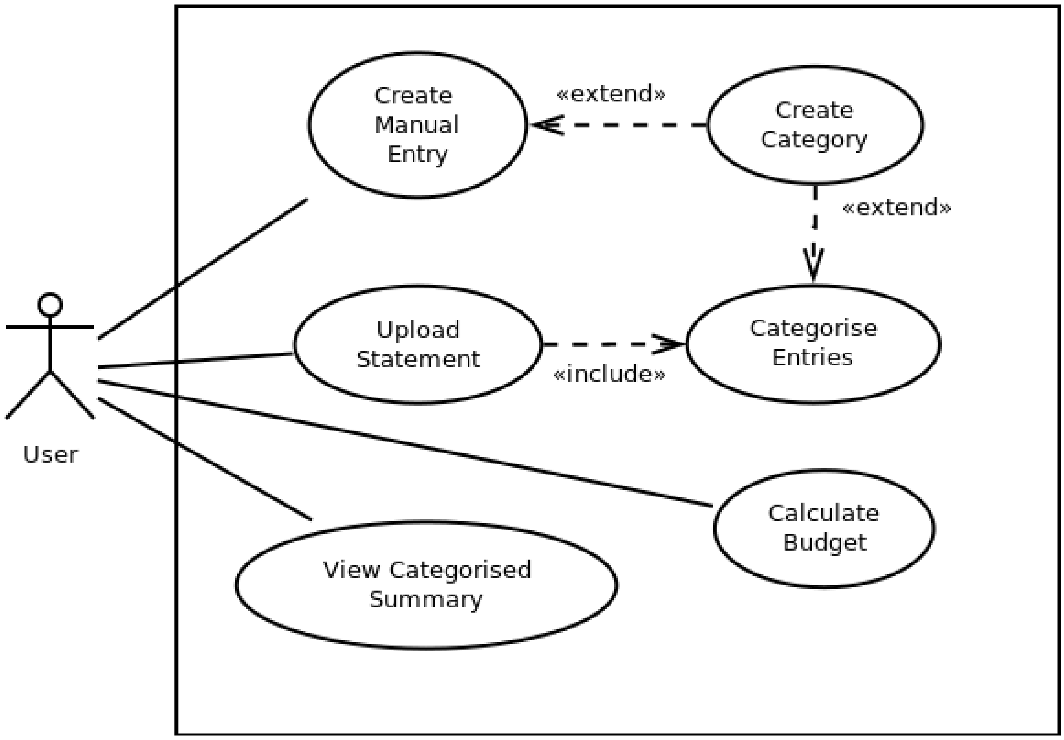


Figure 1: Use Case Diagram

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 either choose an existing category (or categories) for it or create a new one(s) 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
Estimate Tax	Based on the information entered and the current tax year, calculate how much tax is due

Table 1: Use Case Descriptions

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:

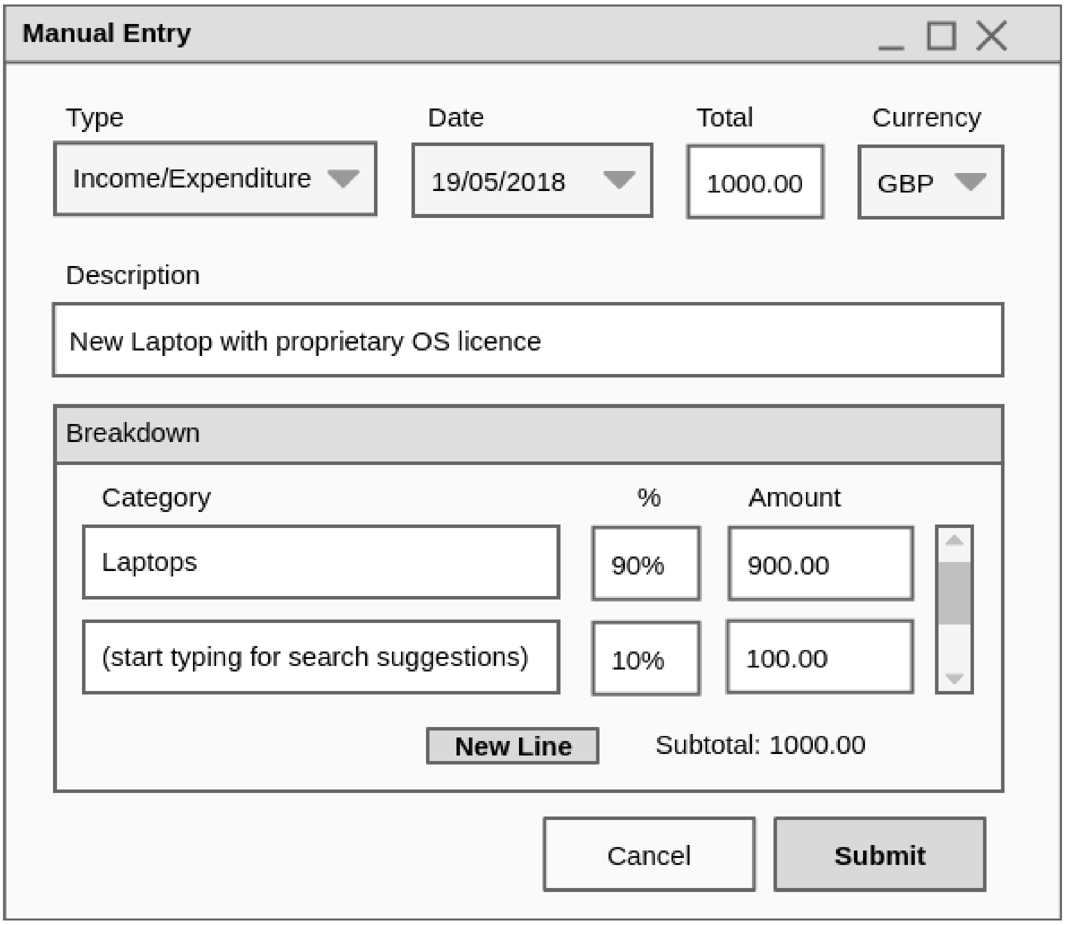


Figure 2: User interface wireframe for *Create Manual Entry* use case

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

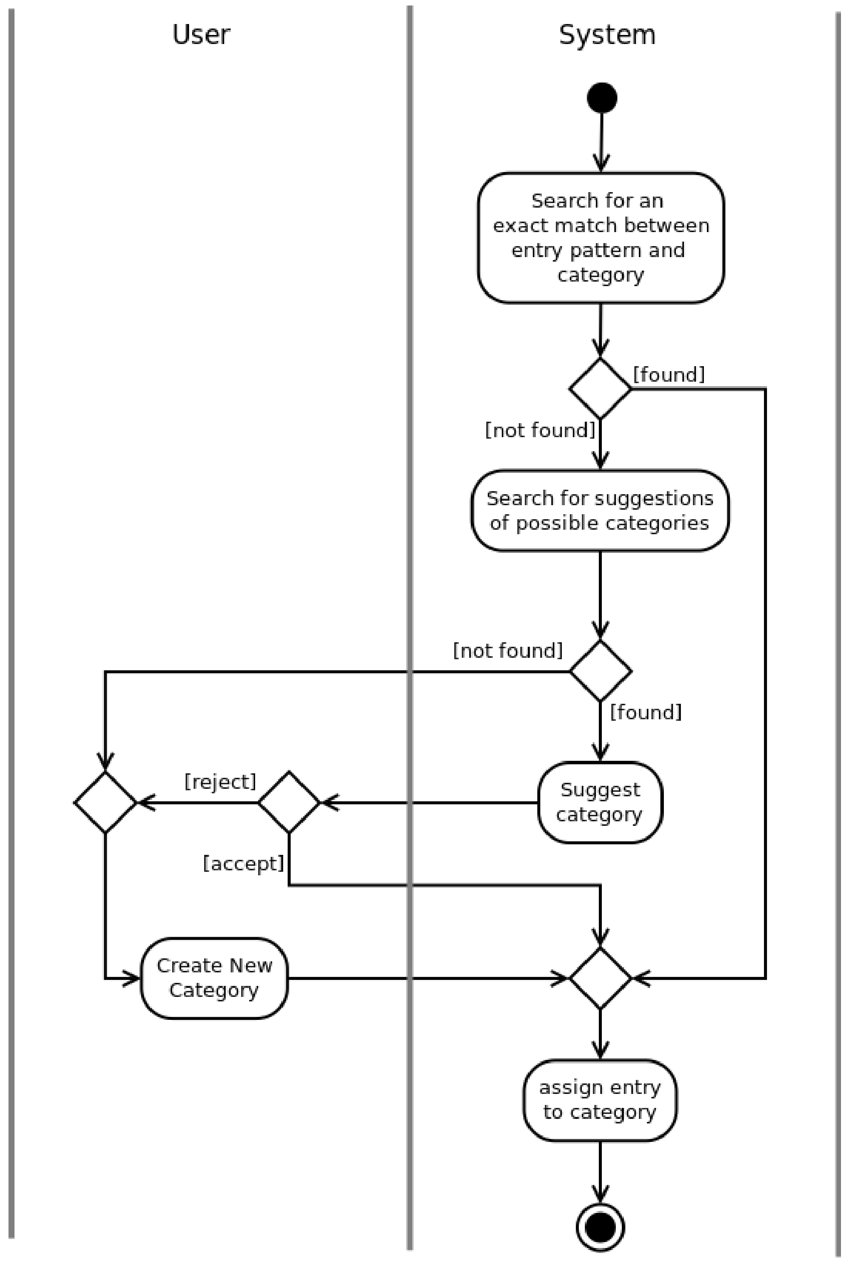


Figure 3

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:

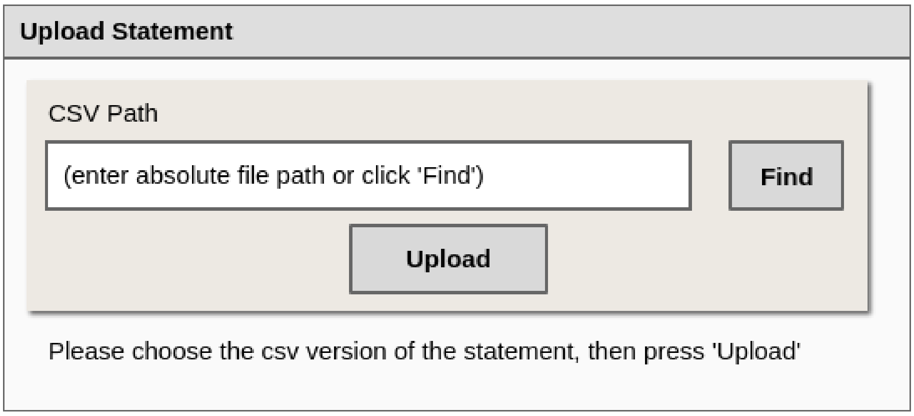


Figure 4

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:

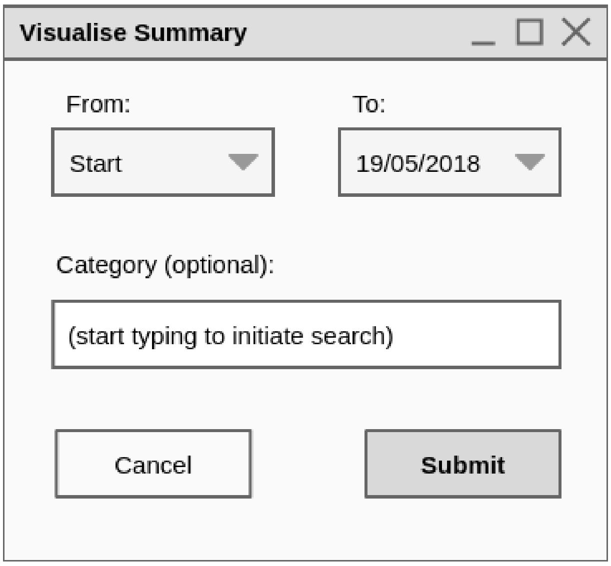


Figure 5

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.

Below is a wireframe to illustrate how the initial window for the *Estimate Tax* requirement:

TODO

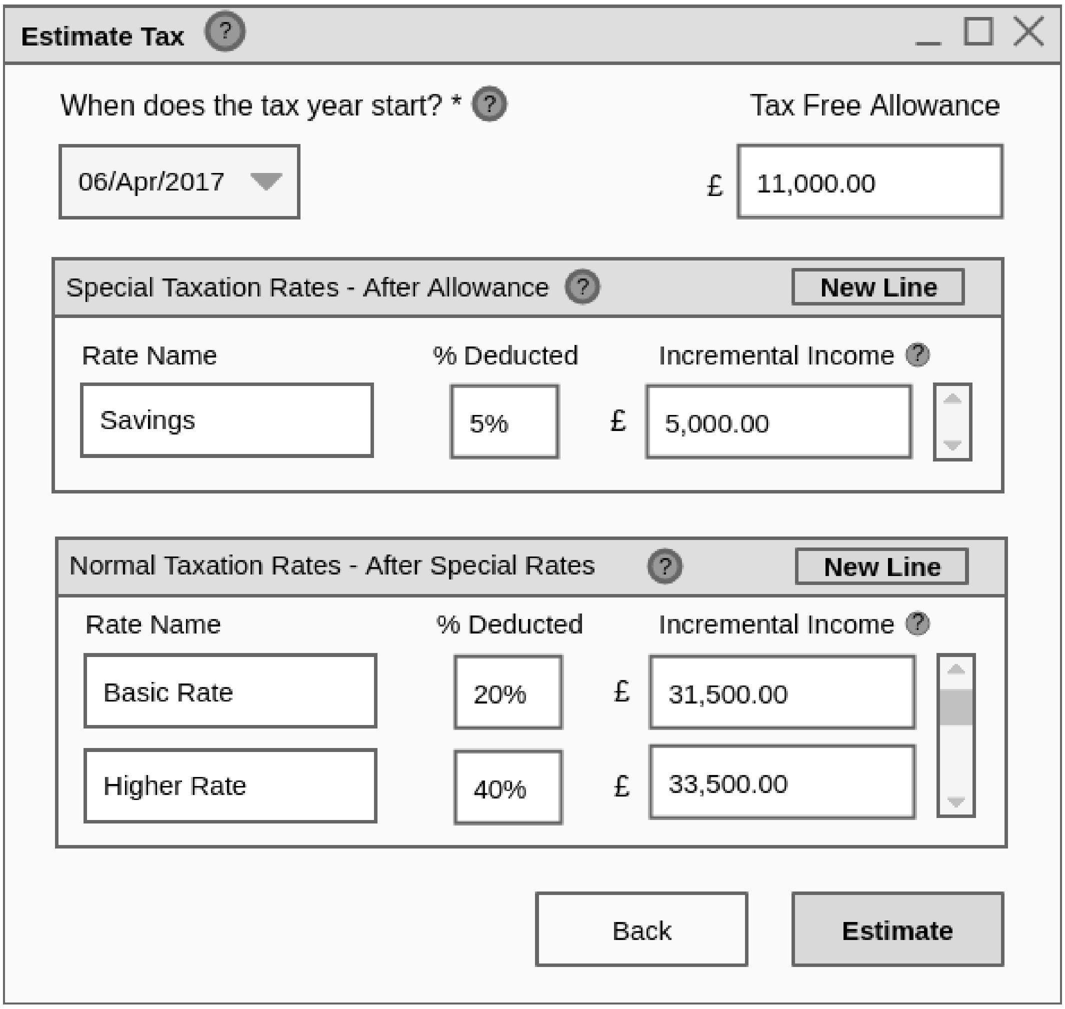


Figure 6: Estimate Tax interface. Date and allowances shown as specified by HM Revenue & Customs (HM Revenue & Customs, 2018)

Normally, in the UK there is a difference between what is "tax exempt" and what is "taxed at 0%". For the purposes of the estimation provided by the requirements in this project, it was initially decided that this distinction would be ignored, and that the user could simply have a choice to use one of the dynamic boxes on *Normal Taxation Rates* to indicate a tax free allowance. However, due to the order in which the taxes are to be deducted, it was decided that a non-taxable allowance field should still be made available, but allow its value to be set to zero. This should allow for more flexibility.

The interface would start with one line item, and then more items could be added if the end user needs them. Each line represents one of the tax rates and the threshold of income which needs to be reached before it starts deducting tax at that percentage.

A choice has been made to include help texts next to the fields which are likely to need them, and these are indicated in Figure 6 by the bubbles with the question marks next to the relevant fields.

3.3 Non-Functional Requirements

Use cases and use case diagrams are an appropriate tool to document functional requirements, but not non-functional ones (Jacobson et al., 1999, cited Bennett et al., 2010, p. 153). Therefore, a separate list has been kept in order to document the non-functional requirements, where they exist.

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).

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 *accounts*. And, in order to make sure to imbue this system with knowledge acquired by more experienced programmers, it makes sense to make use of analysis and design patterns.

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):

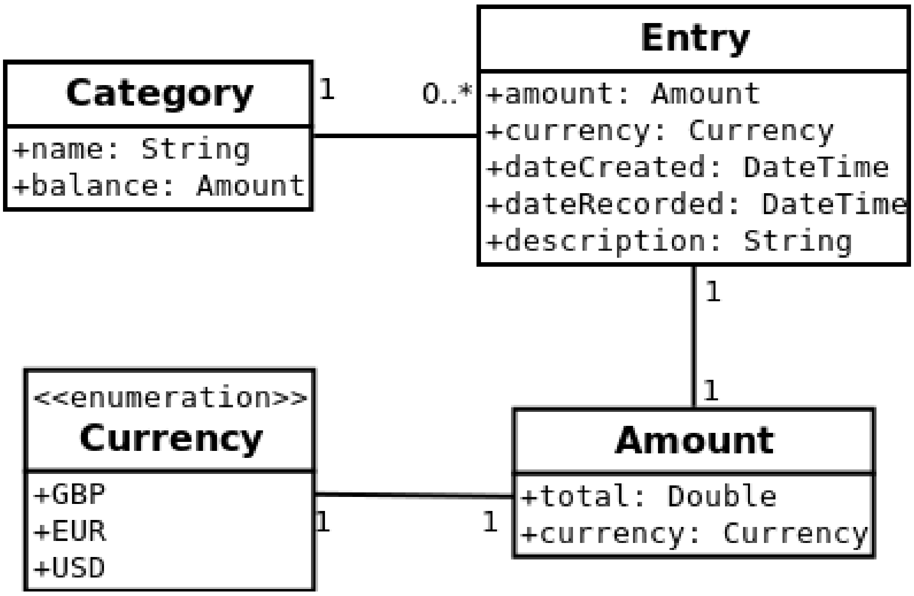


Figure 7

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 17 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 which shall be set at runtime.

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):

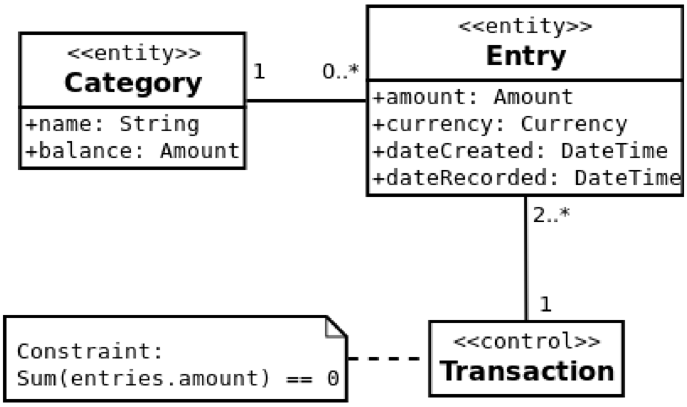


Figure 8

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 10):

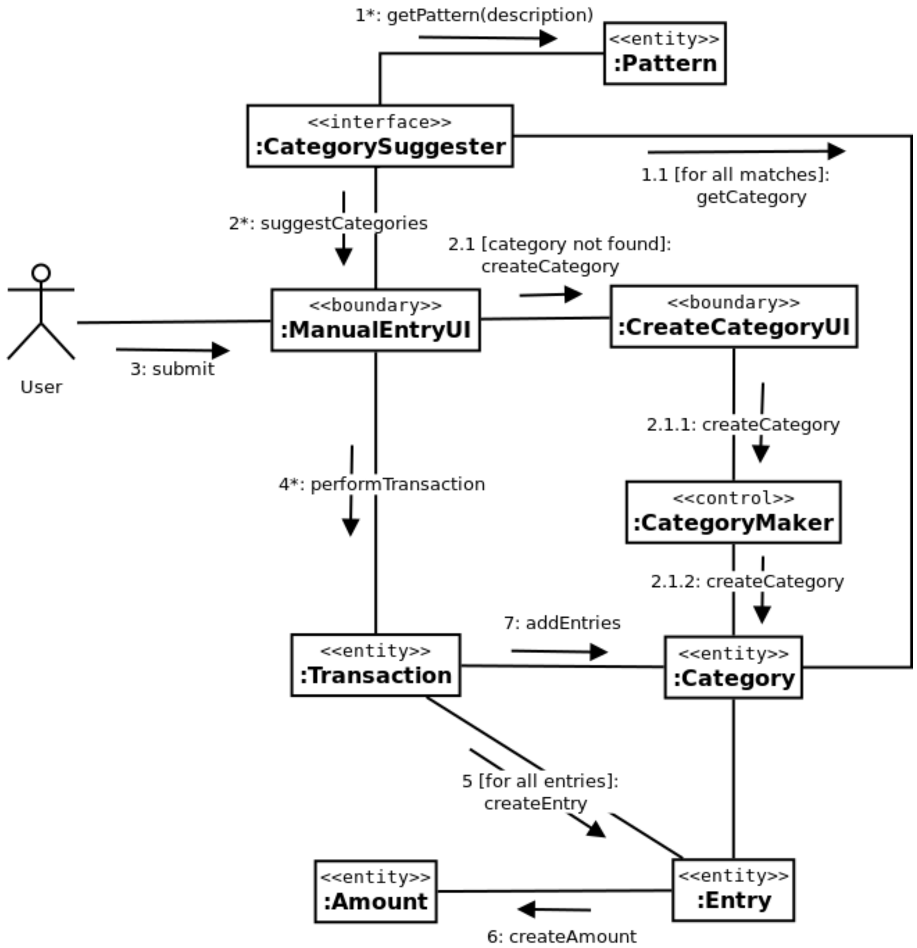


Figure 9: Communication Diagram for the *Create Manual Entry* use case

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 10 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 10) below illustrates this process:

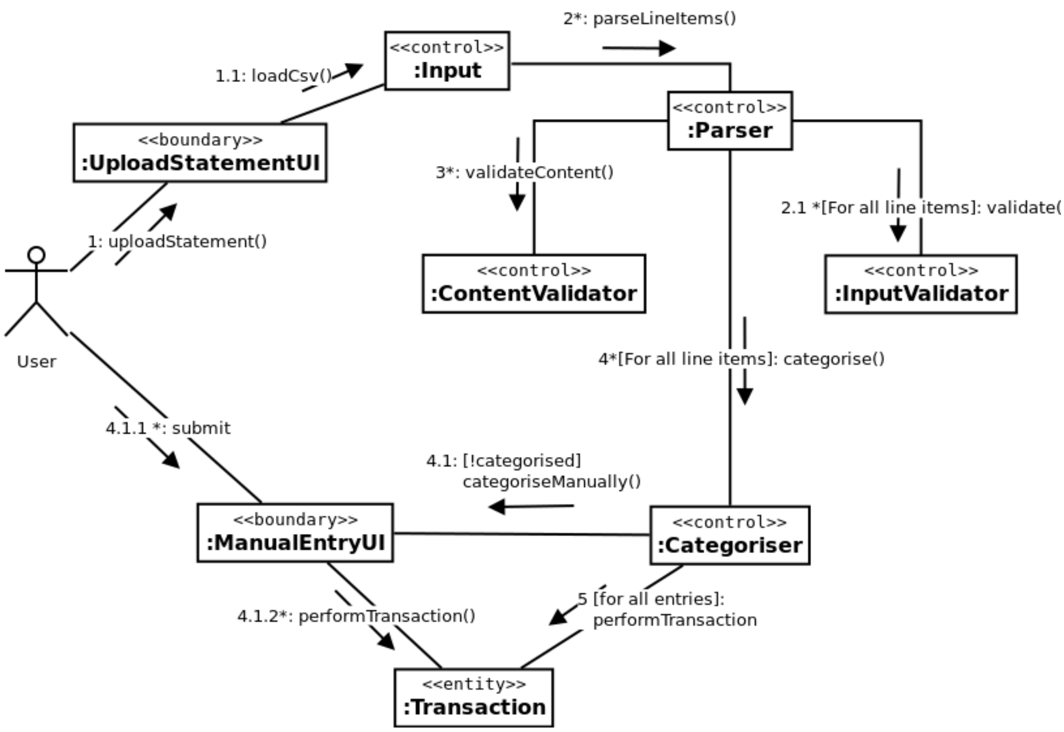


Figure 10: Communication Diagram for the *Upload Statement* use case

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 10.

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 11) is a communication diagram to illustrate this interaction:

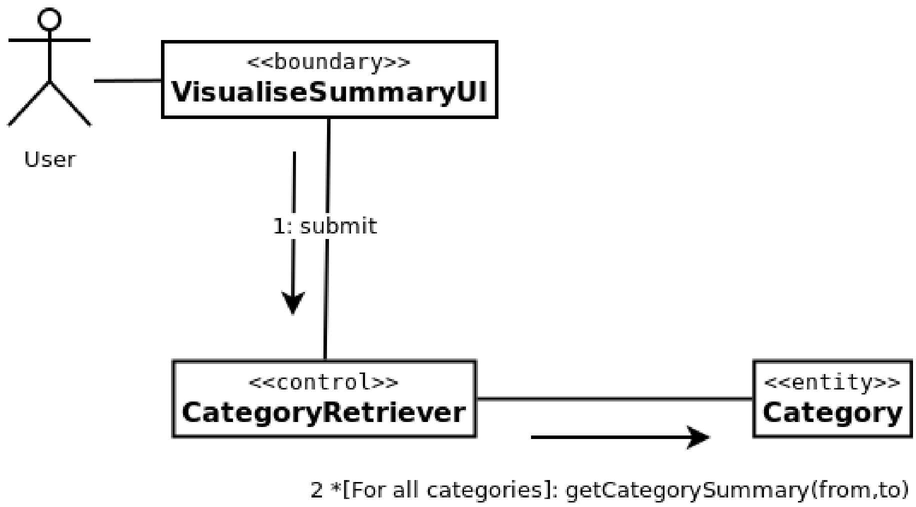


Figure 11: communication diagram for *Visualise Categorised Summary* use case

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 12) 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

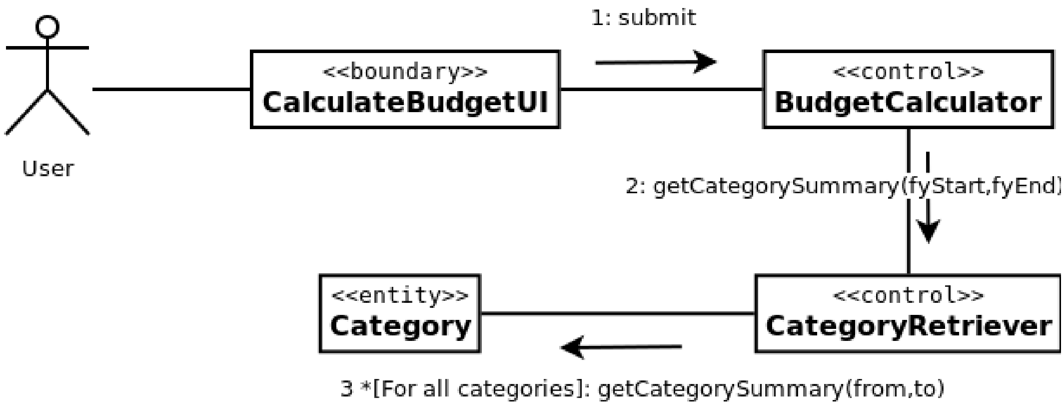


Figure 12: communication diagram for *Calculate Budget*

4.1.5 Estimate Tax

This feature will allow the user to calculate an estimate of the amount of tax due for a financial year, based on their income and expenditure. For this to happen, the user will have to provide the date when the financial year begins, and the tax tiers in their country. The tax estimation will be based as best as possible on how personal tax is calculated in the UK (HM Revenue & Customs et al., 2018).

It was felt that Fowler's (1997, Session. 6.4-6.5) *Memo Account* and *Posting Rules* patterns would be useful for this case. Designing this requirement as such would ensure that, whenever the user created a category which was deemed taxable, a portion of the income would be posted to the tax account. It is also necessary to remember whether income has already been deducted or if it relates to the gross amount, so this will have to be taken into consideration.

The *Memo Account* pattern here will be implemented as tax-related instances of *Category*. Due to the double-entry principle of bookkeeping, each tax instance will need to have a *contra* category – that is, a second category which will act as the counterpart to the transaction – so that the principle is still observed. In order for the user not to have to create the transactions manually, however, an *event listener* needs to be added in order for entries to be made in the appropriate tax categories when necessary. This is where the *Posting Rules* pattern will be appropriate. Below (Figure 13) is an analysis class diagram exemplifying the implementation:

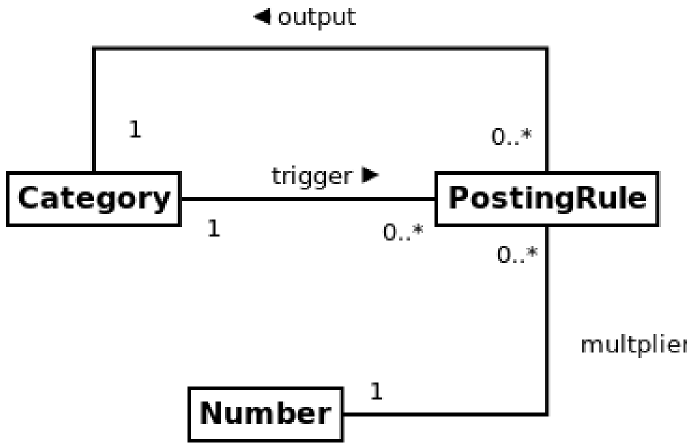


Figure 13

So the principle being illustrated here is that, whenever an entry is posted to a category which the user has marked as subscriber to tax, then the `PostingRule`, which subscribes to that event, would determine whether a transaction needs to happen between the tax category and its contra. Then, when estimates need to be generated, they can simply come directly from the tax categories.

TODO: Regarding designing the tax model Categories should not be aware of tax – in order for the system to truly be developed iteratively, the categories should only keep track of their dates, patterns, etc, and leave the taxation and subscription up to other mechanisms to decide them. The reason for this is that taxes change, but income from previous years may still fall within the previous year's rule. Normally taxes change after tax years have ended, but they may also change within a tax year in some countries, so maybe they should be tied in to the month. Which means that it may be worth thinking more deeply about the class boundaries here – should the tax categories be made aware of dates? Should dates be modelled and from them smaller instances of each system exist – that is, time would be the biggest driver of the domain, so all "rules" would exist within month domains, for example. Then these could be aggregated to form tax years. And still they need to maintain the flexibility to be able to have all entries be queried by whichever period of time the user wants, so ideally the categories should only keep track of their dates, and other mechanisms would have to exist to calculate what subscribes to what. That is, subscription to certain tax rules should exist by period, so ideally the categories should not know about them, but there needs to be something which still can keep track of which category is subscribed to which tax rule at which time. ***TODO WITHIN TODO: when getting to the point of designing and implementing the tax rule***, keep reading fowler's analysis book from section 6.5 – there's an interesting thing about *Method* which may be just the answer to this.

The communication diagram of figure 14 exemplifies the application module which will model this functionality:

TODO

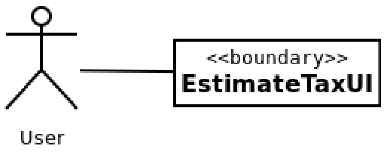


Figure 14

4.1.6 Final Class Diagram

The class diagram on Figure 16 is the result of joining the more relevant entities listed so far:

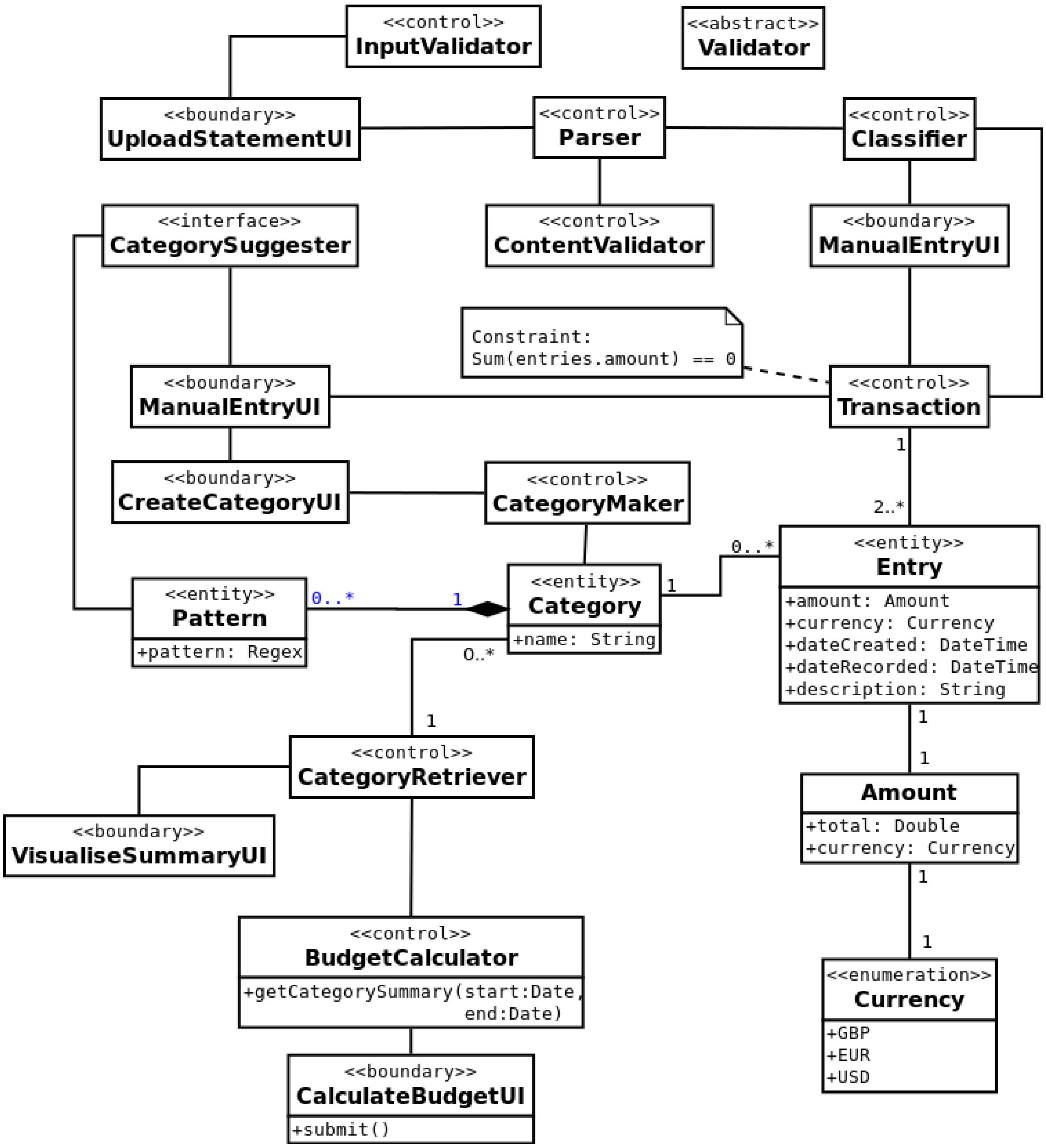


Figure 15: Class diagram of the most relevant classes of the Business Logic Layer

Note that the classes for estimating tax were not included in this iteration, but time permitting they should be included in subsequent ones.

Manual entries and uploaded statements will inevitably always go against a Bank category. For this requirement alone, there does not seem to be any harm in hard-coding the `Bank` category. In this case, since most transactions are going to affect the this category, it was decided that no patterns should be assigned to it. Due to this, and also in order to the *Memo Account* pattern with a *contra* which should be applied to the *Estimate Tax* feature, 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

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

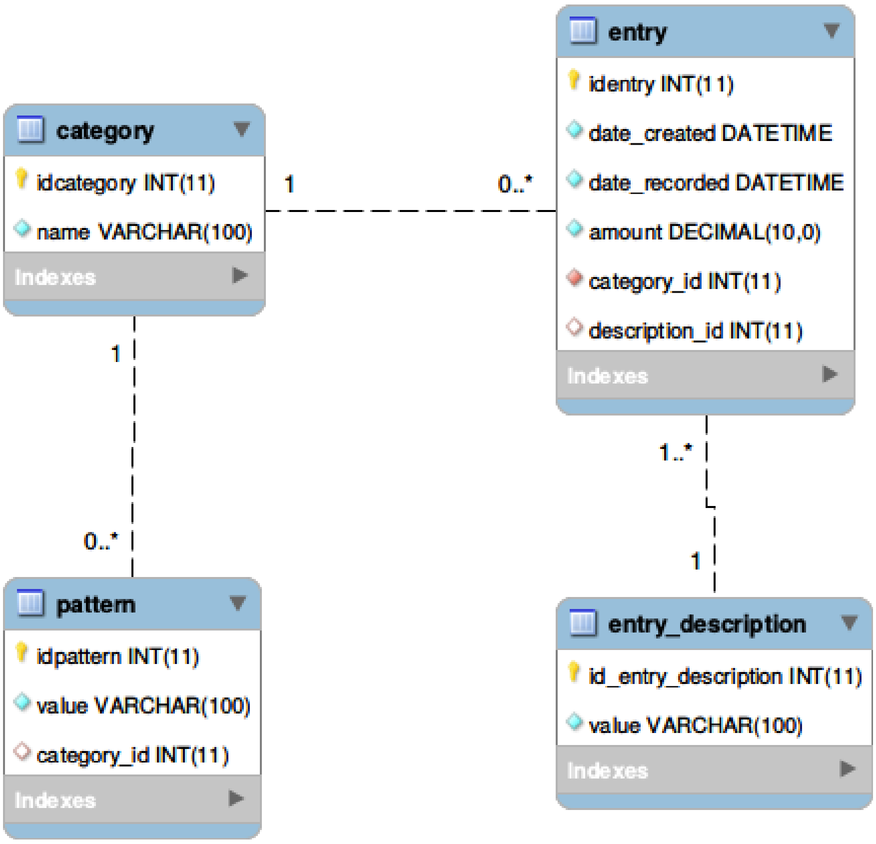


Figure 16: UML Diagram of database schema

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.

5 Implementation

5.1 Scala Case Classes

One of the benefits of Scala are its case classes. They are 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.

Its utilisation 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

```
}

private def addEntries(tu: TransactionUnit): Category = {
  val cat: Category = tu.category
  val ents: List[Entry] = tu.entries
  if (cat.entries.isEmpty) cat.copy(entries = ents)
  else cat.copy(entries = cat.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) 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: write about Scala's own implementation of Mockito as one of the language features

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: List[TransactionUnit]): List[Category] = {
    validators.foreach({vdr: Validator =>
```

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

```
import scala.util.matching.Regex

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

TODO: talk about the choice of ListBuffer for constant append and prepend time (Odersky et al., 2016, location 15415), and List[Map[String,Category]] for Classifier – each map links the same Category to all its patterns, and since Maps have O(1) high probability/almost constant time, it was felt that this would be the best implementation for it. The mapping would work similar to this:

```
Welcome to Scala 2.12.4 (Java HotSpot(TM) 64-Bit
  Server VM, Java 1.8.0_161).
Type in expressions for evaluation. Or try :help.

scala> class K
defined class K

scala> val k = new K
k: K = K@6dc1dc69

scala> val map = Map(1 -> k, 2 -> k)
map: scala.collection.immutable.Map[Int,K] = Map(
  1 -> K@6dc1dc69, 2 -> K@6dc1dc69)

scala> map(1)
res0: K = K@6dc1dc69

scala> map(1) == map(2)
res1: Boolean = true

scala> val map2 = new K
map2: K = K@5fdb7394

scala> val k2 = new K
k2: K = K@70ee1963

scala> val map2 = Map(1 -> k, 2 -> k2)
map2: scala.collection.immutable.Map[Int,K] = Map(
  1 -> K@6dc1dc69, 2 -> K@70ee1963)

scala> map2(1) == map2(2)
res2: Boolean = false
```

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.

6 Testing

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

7 Development Method

For this project, an approach similar to that adopted by Bennett et al. (2010, p. 77) regarding methodology 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 definitions of functional and non-functional requirements from Appendix 1 (9) 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.

7.1 The use of Universal Modelling Language (UML) constructs

UML is a modelling language created with the intention of providing system architects, software engineers and developers with a common set of modelling tools, with a defined syntax, which would help them better analyse and design software-based systems, and to model business and similar processes (OMG, 2015, p. 43). It defines several constructs which have been employed throughout this report in order to model the specifications of the system, such as Use Case, Activity, Class, Sequence and Communication Diagrams.

In the class diagrams used, some thought was given to how to model the relationships between entities. In the end, it was decided that transitory relationships should be modelled as dependencies, and more permanent ones would be modelled as associations. An example can be seen in Figure 18, where the `Category` interface has an association to the `Entry` class, since entries are not going to be removed, only added, to categories, making it a more permanent relationship.

7.2 Requirements Capture Methods

Due to the nature of the system being for personal rather than commercial use – that is, by individuals rather than business entities – the usual fact finding techniques do not apply specifically well. However, the closest match identified to the techniques utilised has been with *'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 qualification 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, so the iterations happened between analysis, design, implementation and testing – that is, requirements were all gathered in a single iteration.

7.3 Analysis and Design

As described by Bennett et al (2010, p. 348), "in projects that follow an iterative lifecycle, 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 an iterative approach, it was decided that the analysis and design of it would be done concurrently.

7.3.1 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).

8 Reflections

8.1 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.

8.2 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 10 and 11 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.

8.3 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.

8.4 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.

8.5 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.

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.

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.

HM Revenue & Customs (2018). *Income Tax rates and allowances: current and past*. URL: https://www.gov.uk/government/publications/rates-and-allowances-income-tax/income-tax-rates-and-allowances-current-and-past (visited on: 03 Mar. 2018).

HM Revenue & Customs, HM Treasury, Department for Work and Pensions, Closed organisation: Department for Business, Enterprise and Regulatory Reform, Closed organisation: Department for Business, Innovation & Skills, Ministry of Housing, Communities & Local Government, Student Loans Company, and The Scottish Government (2018). *Personal tax: Income Tax*. URL: https://www.gov.uk/topic/personal-tax/income-tax (visited on: 03 Mar. 2018).

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.

OMG (2015). *Unified Modeling Language Specification*. Version 2.5. URL: http://www.omg.org/spec/UML/2.5/PDF (visited on: 16 Dec. 2017).

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).

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

9 Appendix I

Bennett et al. (2010, pp. 140-142) categorises requirements as being of three types:

Functional Requirements The system's functionality – what it is expected to do.

Non-functional Requirements How well the system delivers its functionality. These requirements are related to the performance, scalability, availability, recovery time, security, and others.

Usability Requirements These relate to how effectively, efficiently and satisfactorily users can achieve their goals in the existing system. User interfaces can play a big part in meeting these requirements.

10 Appendix II: Previous versions of analysis patterns

Below is a previous version of analysis patterns. It was decided that keeping track of *balance* was not something worth pursuing, since the main point of the application is for users to choose a period of time and then look at a summary of expenditure in there.

The first analysis patterns which seem appropriate are 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):

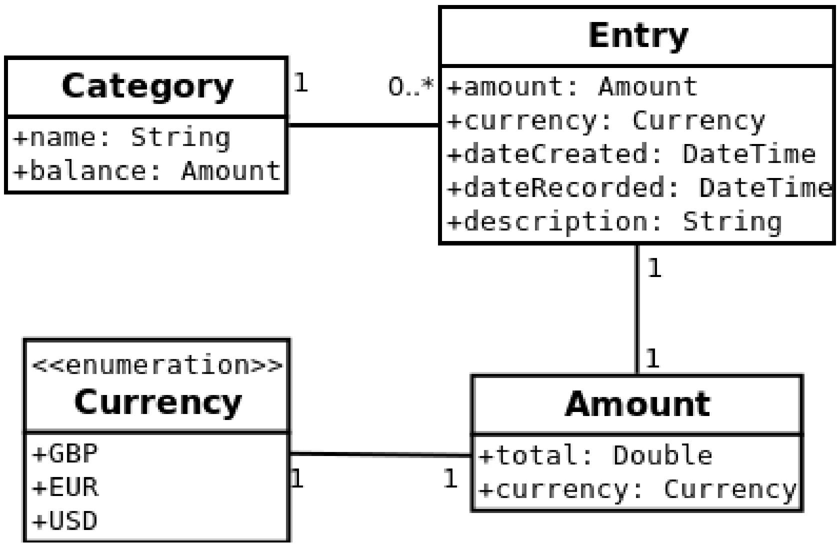
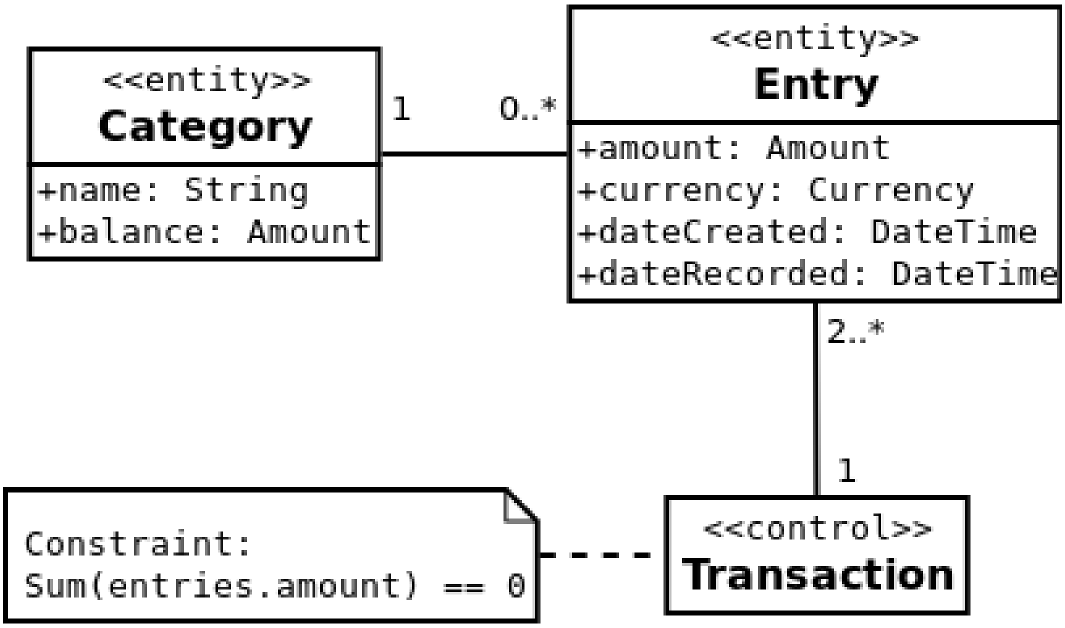


Figure 17

As implied by the diagram above, the *Category* class will be associated with instances of the *Entry* class, and will keep track of the balance made up of the sum of *Amount*'s of each entry. 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.

Another design choice which can be observed in Figure 17 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 design to keep track of transactions in multiple currencies, although it was not a specific requirement. Initially, there will only be a single default currency which shall be set at runtime.

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):



After having determined the analysis patterns which shall be employed, it makes sense to dive into a deeper analysis of the use cases described in Chapter 3. At this point the objective will be to start modelling classes based on concepts or things found in the problem domain. This will be done in the following subsections.

11 Appendix III

The following text was part of a previous model of how to use the *Summary Account* pattern to try to classify accounts between different types. It has been since then decided that the best way would actually to have the categories themselves keep track of their own types.

Furthermore, especially due to the requirements for tax estimates, the *Summary Accounts* pattern (Fowler, 1997, Section 6.3) will be adapted to help classify categories between income and expenditure. At this point, it seemed sensible to make `Category` into an interface, and specialise it on its subtypes. A `SummaryCategory` implements the `Category` interface, and would implement its `getEntries` method so that its entries are those of its components. It's components are other instances of `Category`, so they can be both both `DetailCategory` and its own type – the implementation will have to take this into account somehow, such as by using recursion. The class diagram below (Figure 18) can illustrate it better:

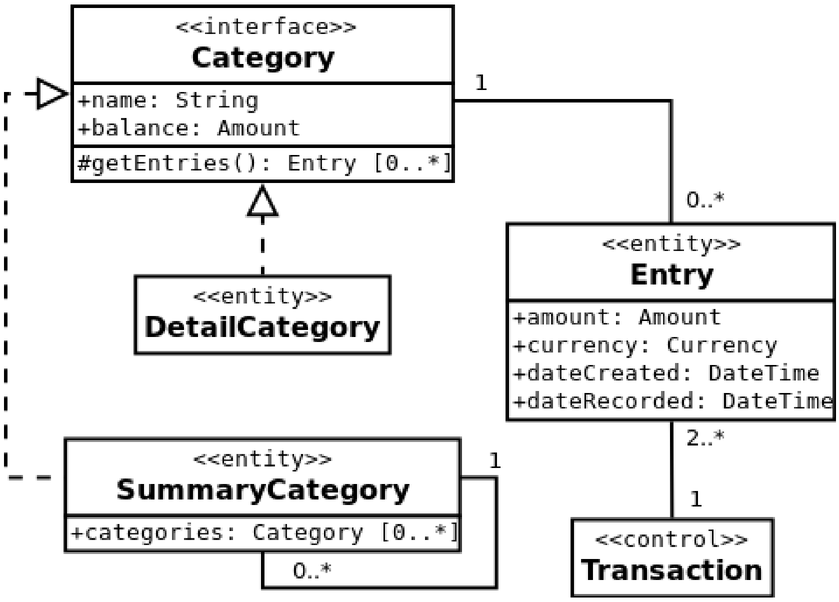


Figure 18