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

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 these ideas.

The system 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* acccounts (Wood et al., 2004, pp. 18-19).

This report documents the work of the project. Each chapter delineates a specific aspect of the development lifecycle, which is in line with the development process listed in 3. Chapter 4 identifies the identified requirements which were used as motivation for the system to be developed.

Analysis of requirements has been incorporated in the design section (Chapter 5). This is also where the first mention of patterns can be seen.

3 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.

3.1 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 diagrams As a useful, high level tool to document users' requirements (Bennett et al., 2010, p. 138), use case diagrams have been used to develop the requirements model of the system.

Activity diagrams

Class diagrams

Sequence diagrams

3.2 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 capture the main requirements.

3.3 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.

3.3.1 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).

4 Requirements

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.

The definitions of functional and non-functional requirements from Appendix 1 (7) 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.

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.

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.

4.1 Functional Requirements

4.1.1 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:

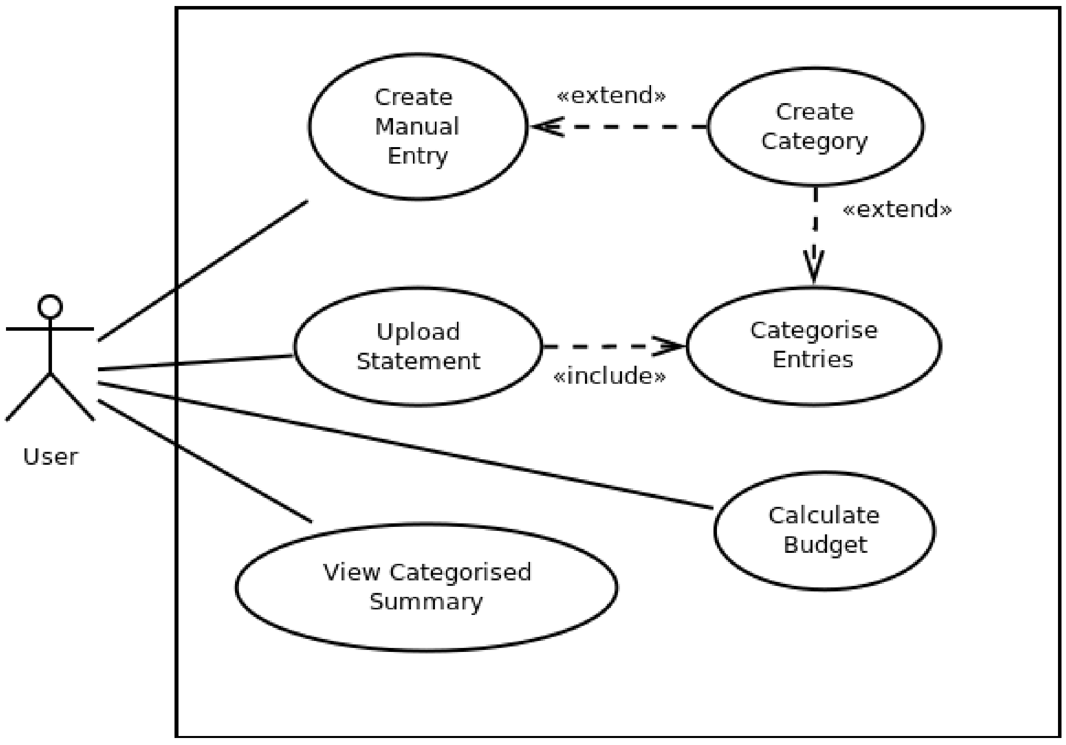


Figure 1: Use Case Diagram

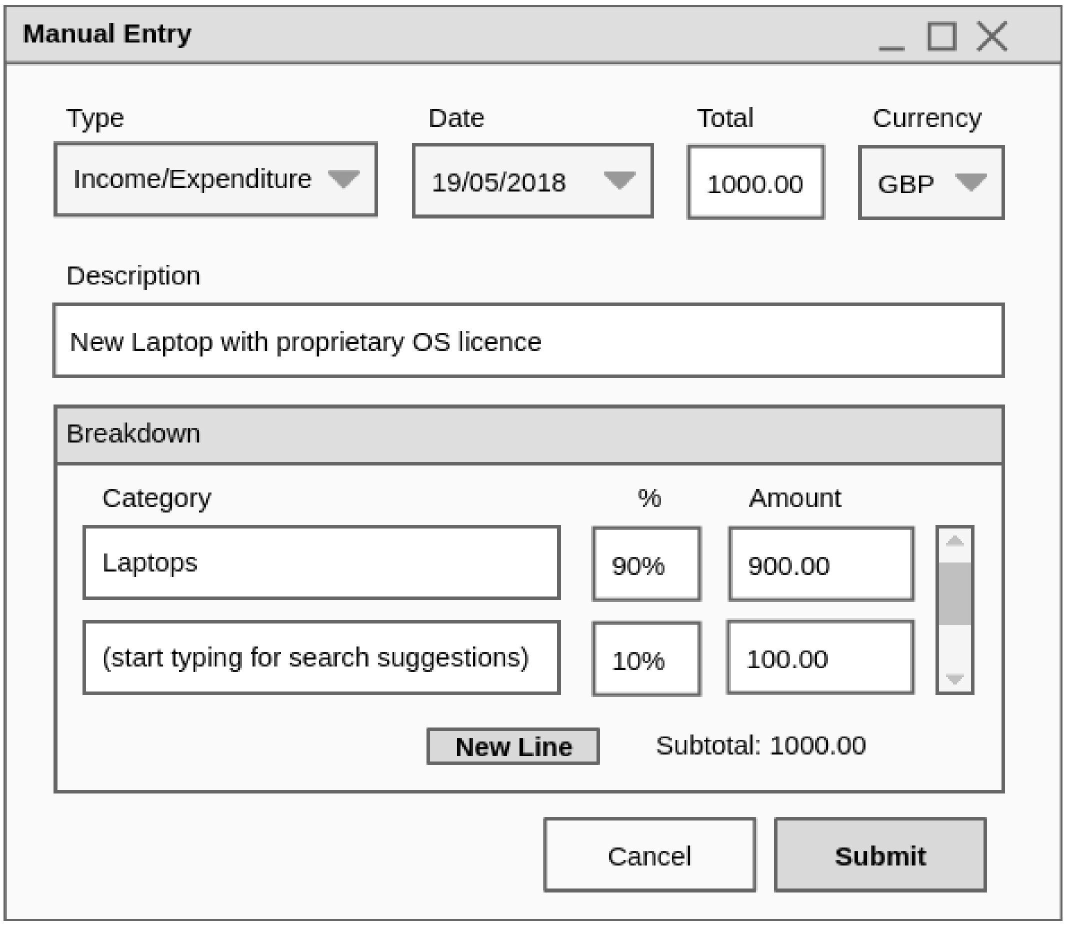
4.1.2 Use Case List

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 entry for income or expenditure, include a date, amount and description, and either choose an existing category for it or create a new one in the process *Extends*: Create Category
Visualise Categorised Summary	The use 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

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:


Manual Entry

Type	Date	Total	Currency
Income/Expenditure	19/05/2018	1000.00	GBP

Description: New Laptop with proprietary OS licence

Breakdown

Category	%	Amount
Laptops	90%	900.00
(start typing for search suggestions)	10%	100.00

Subtotal: 1000.00

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:

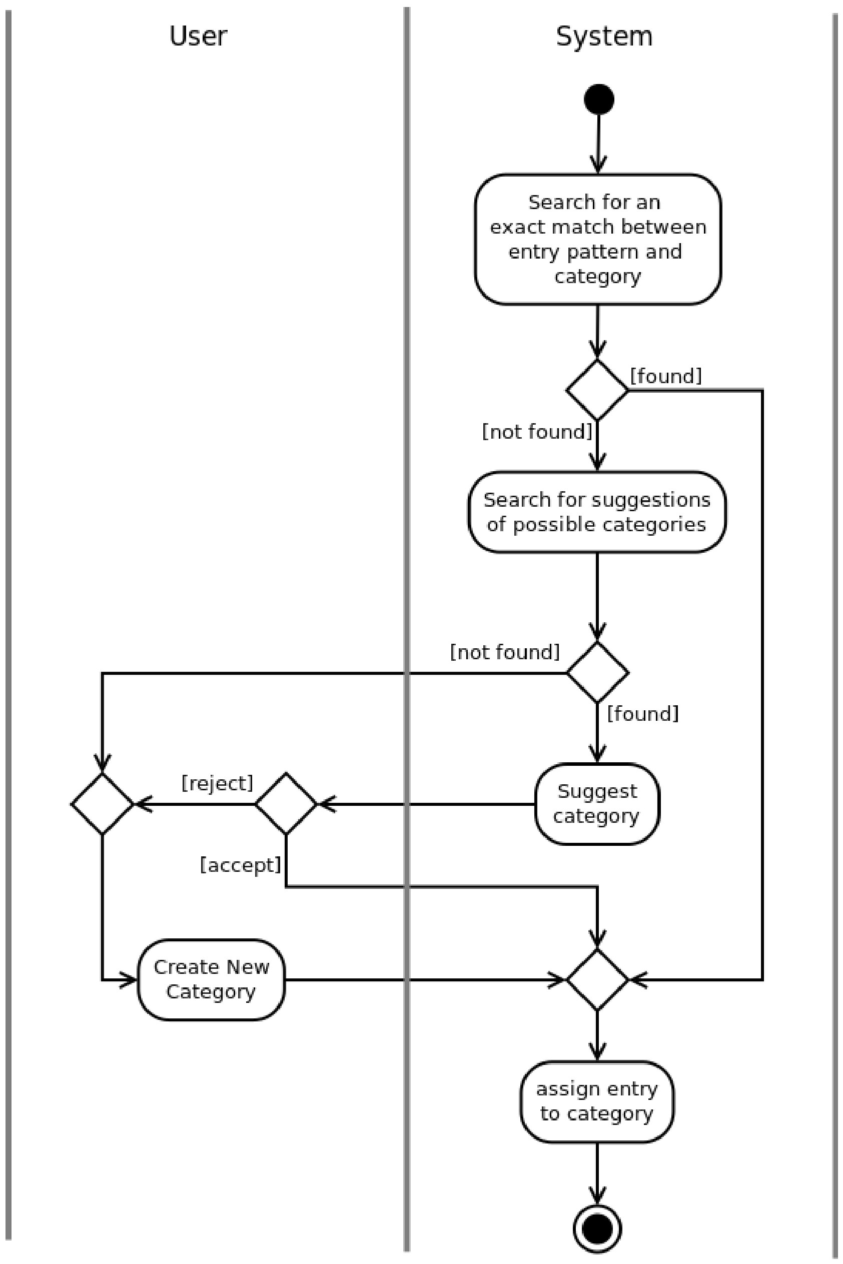


Figure 3

Below (Figure 4) 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:

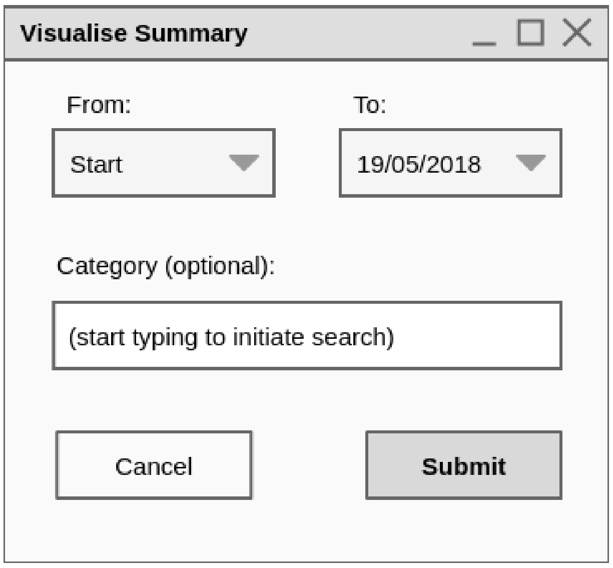


Figure 4

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.

4.2 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.

5 Analysis and Design

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):

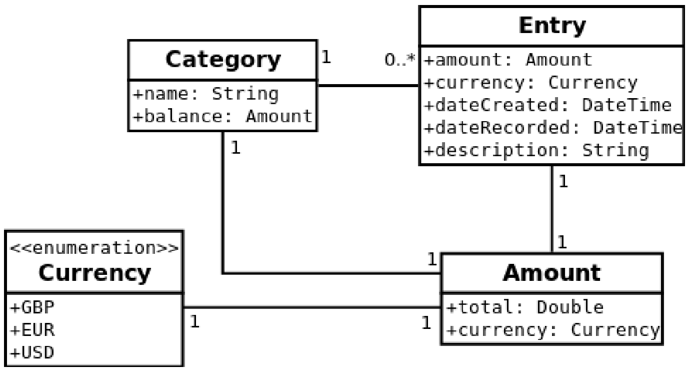


Figure 5

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. The modification 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 5 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):

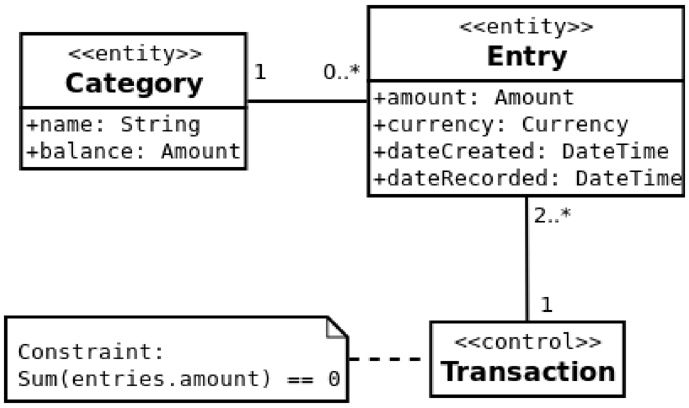


Figure 6

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. 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 7) can illustrate it better:

TODO

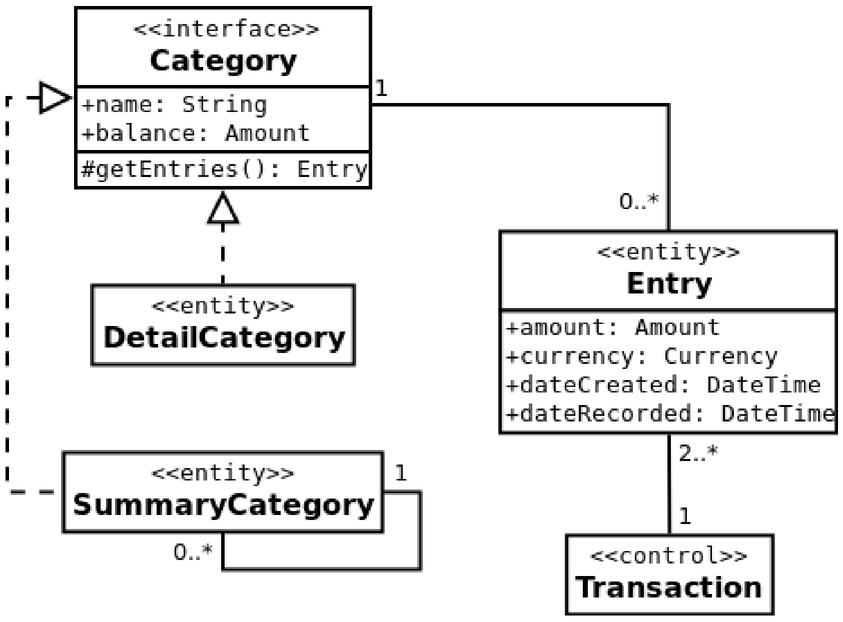


Figure 7

Lastly, also as a preparation for the estimate tax functionality, it would be useful to design them based on the *Memo Account* and *Posting Rules* patterns (Fowler, 1997, Section 6.4 & 6.5). These patterns, however, will have to be slightly modified in order to better suit the design of this system. For example, in Fowler's original implementation, a *Memo Account* is a subtype of *Account*, but it's not subject to double entry – that is, whenever a double entry is made there can be also a third entry in the memo account. In this system, the categories will be subject to double entry, and when a percentage is registered in the memo category it will not be entered anywhere else. **TODO: revise this. is it really a good idea? Memo accounts are subsidiary ledgers, so maybe it would be good to keep them as such...?**

TODO: implement the Memo Account and Posting Rules, and employ textual descriptions

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 4. 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.

5.1 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):

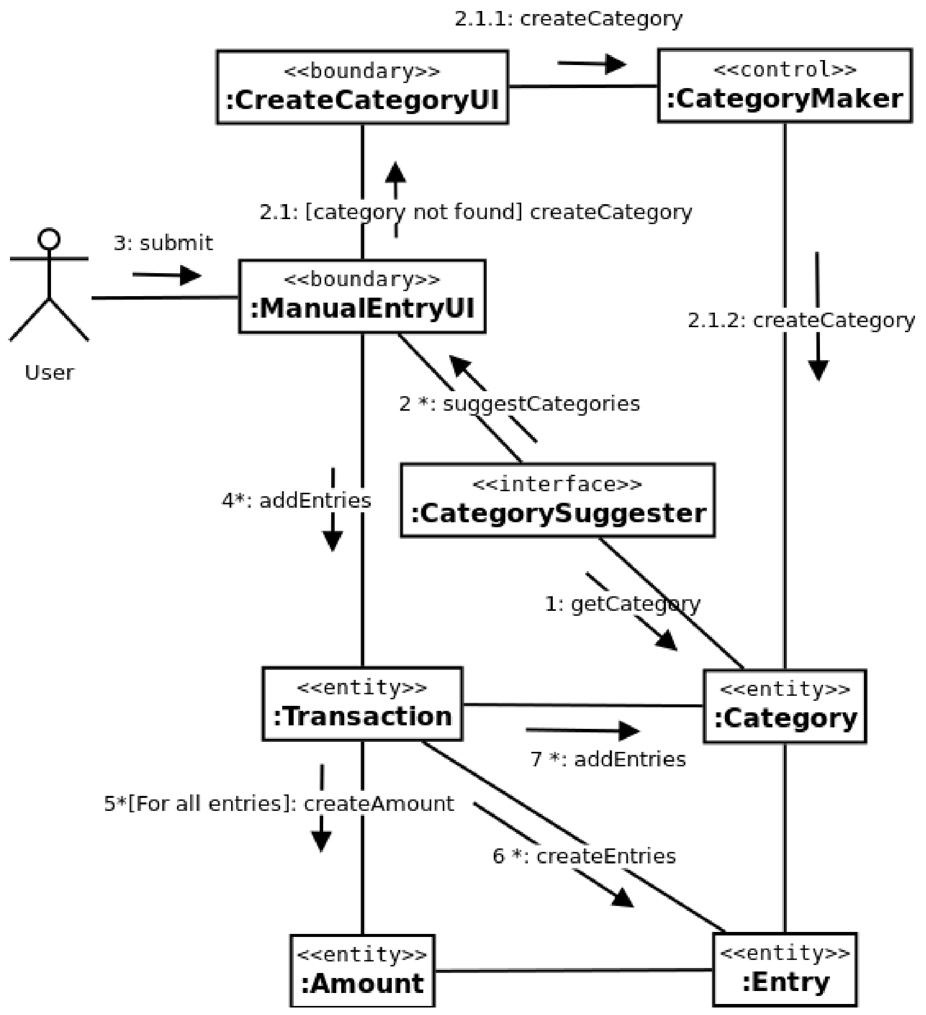


Figure 8: Communication Diagram for the *Create Manual Entry* use case

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.

5.2 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:

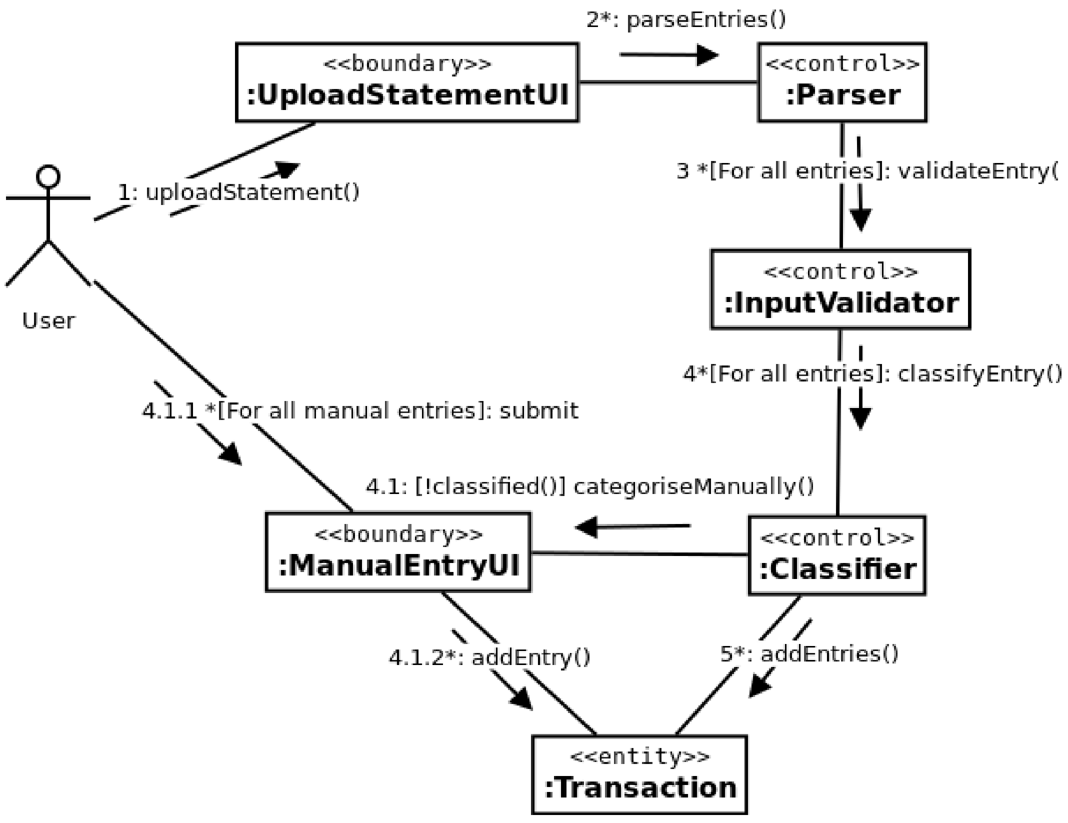


Figure 9: Communication Diagram for the *Upload Statement* use case

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

5.3 Visualise Categorised Summary

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

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

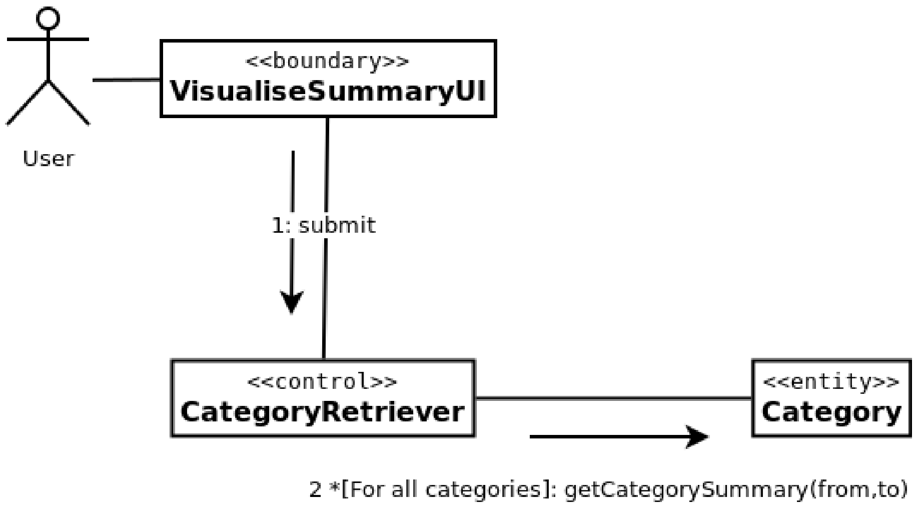


Figure 10: communication diagram for *Visualise Categorised Summary* use case

5.4 Calculate Budget

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

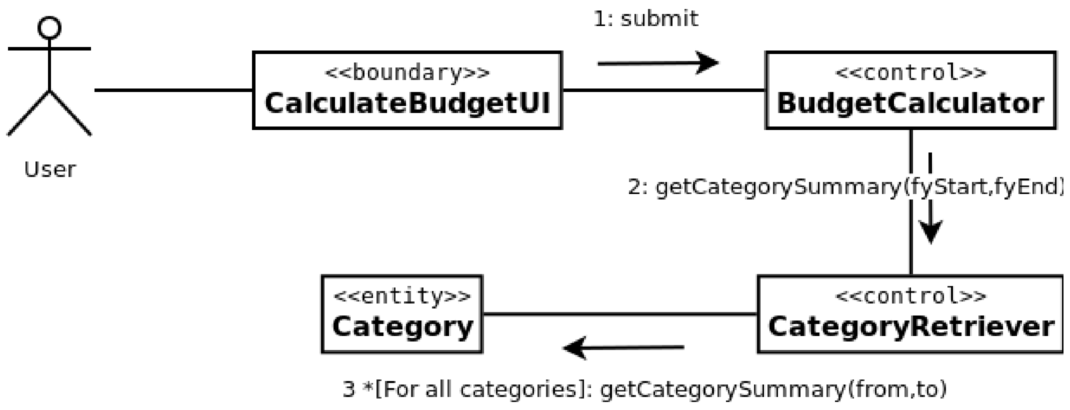


Figure 11: communication diagram for *Calculate Budget*

5.5 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. This is loosely based on UK Generally Accepted Accounting Practices (UK GAAP), where taxation is centralised and happens in tiers. The communication diagram of figure 12 exemplifies the application module which will model this functionality:

TODO

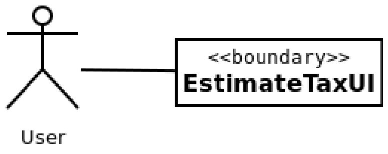


Figure 12

6 Reflections

6.1 Expert Systems

Originally, the author did not know about expert systems when the idea for this project was conceived. However, during the literature search and review, the idea for these systems was found, and many of the patterns of what an expert system does and what this system is supposed to do were identified to be similar. This led to the conclusion that the project has the potential to become an expert system, even if just with a budgeting tool. The expert knowledge being provided by it for its first iteration includes:

- Double entry bookkeeping;
- Budgeting by category.

It achieves the above by separating the inference engine, which is the tool responsible for knowing how to apply double entry to transactions, from its knowledge base which is the information input by the user – for example, if the user tells the system a manual entry is income, the system will know to debit the cash book, and credit the category in question.

Brown et al. (1990) declare that the heuristics used by a financial planning system can be interpreted as a "rule of thumb" to be applied to a problem which will normally result in a correct solution for it. In the same article it is also stated that "an expert system is most commonly and most effectively used as an advisor to a human decision maker". I this is considered as the measure by which classify an expert system, then the budgeting tool alone would place this system into it.

6.2 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.

References

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.

Brown, Carol E, Nielson, Norma I, and Phillips, Mary Ellen (1990). "EXPERT SYSTEMS FOR PERSONAL FINANCIAL PLANNING." In: *Journal of Financial Planning* 3.3.

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.

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.

7 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.