Buddi

## Waseem Abed 6699286

## Azin Zareian 6918808

## Ajay Kumar Thakur 7138245

## Yugant Sobti 7143389

## Arash Abolfathi 7135122

Due: Mar 3th, 11 marks

**Our repository:** [**https://github.com/waseem000/SOEN-6471-Buddi**](https://github.com/waseem000/SOEN-6471-Buddi)

# Summary of Project

Buddi is an application that helps people with little or no financial background to manage their finances. The application allow the users to add different types of accounts, record the transactions between the Accounts and it allow them to generate different types of budgeting reports such as graphical reports and textual reports. The application was built in Java so it is self-contained and it runs on Windows, Macintosh OS X and Linux environments, as it simply requiring Java virtual machine to be installed. The Buddi application is available in multiple languages and it was released under GNU General Public License. We choose this particular project because of the rich functionality that the application offers, and the domain, which is financial management, is something that we all can relate to and benefit from.

# Class Diagram of Actual System

## Ideal architecture from M2

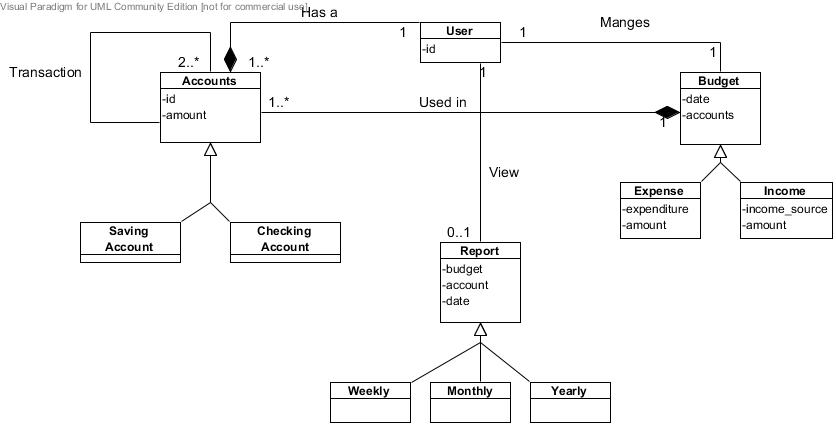


Figure : Ideal architecture

## Class Diagram of Actual System

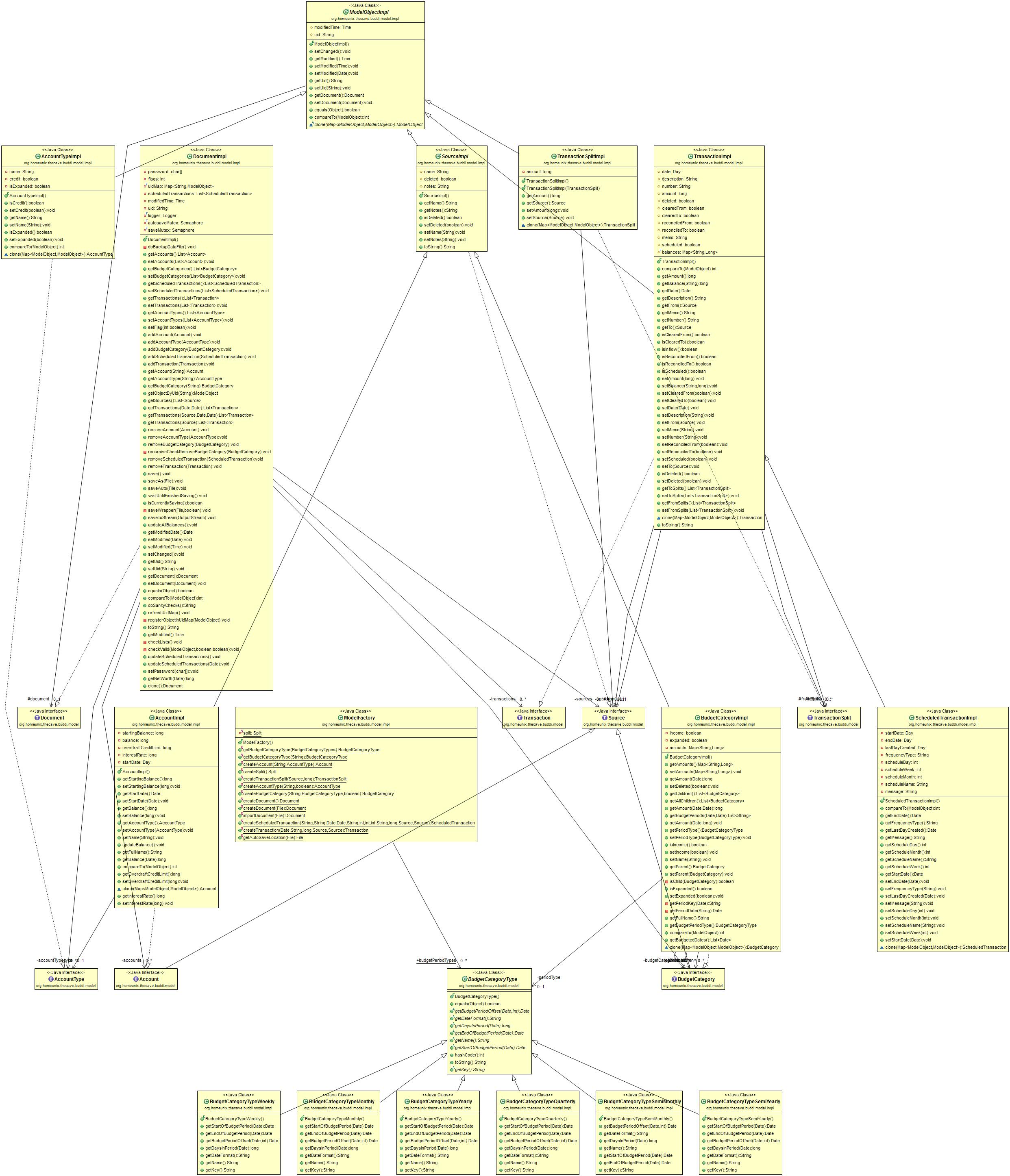


Figure : Class Diagram of Actual System

The Domain diagram illustrates the main conceptual classes in the project. Although we have not considered some details so that the diagram could fit properly; but still the diagram shown depicts all the necessary details and fundamentals about the project.

Buddi is organized as a three-layered architecture (Figure 3):



Figure Three layered architecture

The diagram depicted above is somewhat organized in the same manner.

The class, which corresponds to the business logic, is the Account class in relation with the budget and transaction. Communication across packages will be done by the means of interfaces. Objects from the user interface communicate with the entities and which in turn makes a communication with the database. The combination of all the below listed techniques enabled us to validate and also allowed us in adjustment of some of the assumptions made by us. BUDDI is a complex application, and the code consists of many powerful object-oriented techniques such as information expert, polymorphism generics and role-based interfaces.

We used a few different techniques in order to help build our model:

1. Reverse engineering tool is the Object Aid that helps in visualizing the dependencies amongst the classes and their hierarchal pattern.
2. We manually analyzed the classes using the eclipse platform.
3. For analysis of code and finding the bugs using “FIND BUGS”1.

**Variations**:

There are some discrepancies between the actual architecture of the BUDDI and the architecture that we suggested in the milestone 2 previously. We were able top find out most of the classes and their interdependence amongst each other but somehow some of the following conceptual aspects were not acknowledged by us:

**User:** we thought that a “User” is a conceptual class playing the central role in the domain and it was positioned it in such a way that it was connected to all the major concepts like budget, reports, Transaction, Accounts etc. but the actuality was that it is a single user application.

**Budget Category**: We assumed in our architecture that the budget class has two types of budgets and that were the income budget (contains all the sources of income like salary, bonus, incentives etc.) and the expense budget (it contained the aspects where the user could spend the amount like household, bills, hospital, education etc.) but in real domain diagram there was not such types of budgets under budget class but these (income and expense budget) were the classes itself.

**Report**: The architecture suggested by us contained a report class which would be viewed by the user in order to look at his selected budget (monthly or weekly) but in the actual domain the report is depicted by document implementation class that is related to account and not budget as we proposed.

**Two Classes and Their Relationships:**

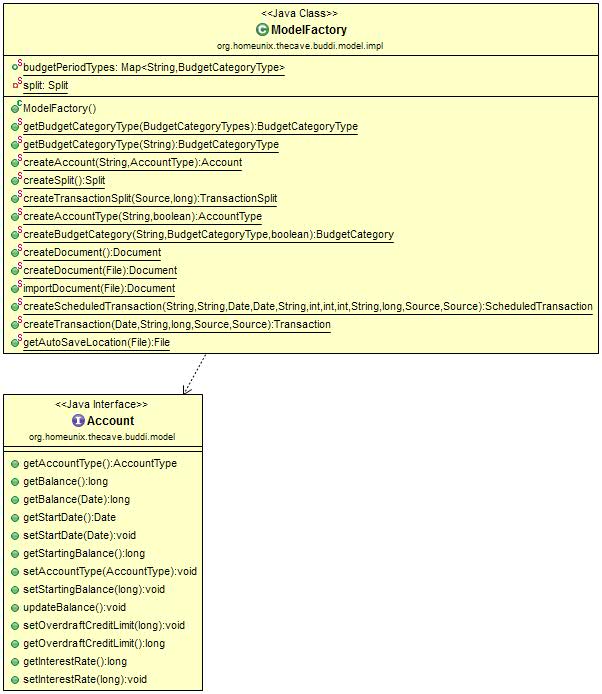


Figure : Sample classes

ModelFactory class is placed under org.homeunix.thecave.budddi.model.Impl. It is the factory for all model objects and as per the architecture this class is highly recommended for the use instead of using the constructors for all model objects like Account Type, Budget Category, Transaction Split.

With the help of this class we can create Account interface which in turn creates a new account with the parameters like name of the user, type of the account to be created etc.

Code Snippets:

public class ModelFactory   
{

public static Map<String, BudgetCategoryType> budgetPeriodTypes;

public static Account createAccount(String name, AccountType type) throws InvalidValueException {

Account a = new AccountImpl();

a.setName(name);

a.setAccountType(type);

return a;

}

}

public interface Account extends Source {

public void setAccountType (AccountType accountType) throws InvalidValueException;

}

# Code Smells and System Level Refactorings

Figure5 shows Accounting system architecture the view layer presents user interface which contains business logic to create, read, update and delete operations (CRUD). In addition, it provides reports and presents them to the user. Model contains classes, which have different responsibilities; they hold data and implement part of business logics for some CRUD operation. The model layer has a direct communication with database file and commits changes. Furthermore, it imports collected data to database. This type of architecture leads system to have high coupling and low cohesion. Each object has different responsibilities, which it can be import data to database or present information in user interface. Therefore, it is hard to track impact of changes in software and extend the software to add new features.



Figure 5: Accounting system architecture

## Code smells signs:

* Large classes, class “DocumentImpl” contains 1317 lines of code and it is responsible for CRUD operation for all domain class in database file.  Moreover, it retrieves all information from file by running software.
* Instance Message chains, in class “BudgetCategoryImpl”. There is this function call statement to find period Date getBudgetPeriodType().getStartOfBudgetPeriod(DateUtil.getDate(year,month, day));
* Long Method, in class “TransactionFrame”. There is actionPerformed method with almost 224 lines of code.

To decrease coupling and create single responsible object we try to find a way to separate domain from presentation for this purpose we made some changes in architect of software. Then, we apply MVC pattern to project it add controller classes (Figure 6). The controller defines application behavior, map user action to model updates and select views to present. So, we can move many methods to controller from view classes and it help us to separate business from presentation.



Figure 6:MVC Pattern

After applying MVC pattern to project, we still need to find a way to modify our large classes in model layer to make them less complex, understandable and manageable. Therefore, we try to use some principals from Domain driven design architecture to encapsulate complex business logic that help us to extend business logic easily. We add service layer in top of Model and move business functionality to them. Then, we change the model to be as a plain old java object and they act just as a data holder (Figure 7).



Figure 7 : Adding service layer

During this operation and changes in software architecture we applied these refactoring principles:

* Extract class, each class in model layer changes to two separate classes. Model which will act as a data holder and service class to implement business logics. For example, TransactionImpl will change to two separate classes, Transaction and TransactionService (Figure 8).



Figure 8 : Extract TransactionImp

* Move Field, fields related to Transaction moves from TransactionImp to Transaction Class,

String description;

String number;

**long** amount;

Source from;

Source to;

**boolean** deleted;

**boolean** clearedFrom;

**boolean** clearedTo;

**boolean** reconciledFrom;

**boolean** reconciledTo;

* Move Method, all behavior associated with the TransactionImp will need to be moved to TransactionService. For example, we remove method getBalance from TransactionImp to TransactionService.

**public** **long** getBalance(String sourceUid)

* Extract Method, to make methods more simple and clear. For example, setClearedFrom(boolean cleared) changes to two different methods setClearedFrom(boolean cleared) and create new method called IsToClear(Transaction transaction)

**public** **void** setClearedFrom(**boolean** cleared) {

**if** (**this**.clearedFrom != cleared)

setChanged();

**this**.clearedFrom = cleared;

**if** (**this**.getTo() != **null**

&& **this**.getFrom() != **null**

&& !(**this**.getFrom() **instanceof** Split)

&& !(**this**.getTo() **instanceof** Split)

&& (**this**.getFrom() **instanceof** Account || **this**.getFrom() **instanceof** BudgetCategory)

&& (**this**.getTo() **instanceof** Account || **this**.getTo() **instanceof** BudgetCategory)

&& (**this**.getTo() **instanceof** BudgetCategory|| **this**.getFrom() **instanceof** BudgetCategory|| ((Account) **this**.getTo()).getAccountType().getName().equals(TextFormatter.*getTranslation*(BuddiKeys.*PREPAID\_ACCOUNT*))|| ((Account) **this**.getFrom()).getAccountType().getName().equals(TextFormatter.*getTranslation*(BuddiKeys.*PREPAID\_ACCOUNT*)))){

**this**.clearedTo = cleared;

}

}

will be changed to

**public** **void** setClearedFrom (**boolean** cleared) {

**if** (**this**.clearedFrom != cleared)

setChanged();

**this**.clearedFrom = cleared;

**this**.clearedTo = IsToClear(**this**);

}

}

**public** **boolean** IsToClear(**TransactionService** transaction) {

**return** (transaction.getTo() != **null**

&& transaction.getFrom() != **null**

&& !( transaction.getFrom() **instanceof** Split)

&& !( transaction.getTo() **instanceof** Split)

&& (transaction.getFrom() **instanceof** Account || transaction.getFrom() **instanceof** BudgetCategory)

&& (transaction.getTo() **instanceof** Account || transaction.getTo() **instanceof** BudgetCategory)

&& (transaction.getTo() **instanceof** BudgetCategory|| transaction.getFrom() **instanceof** BudgetCategory|| ((Account) **this**.getTo()).getAccountType().getName().equals(TextFormatter.*getTranslation*(BuddiKeys.*PREPAID\_ACCOUNT*))|| ((Account) **this**.getFrom()).getAccountType().getName().equals(TextFormatter.*getTranslation*(BuddiKeys.*PREPAID\_ACCOUNT*))));

}

* Introduce Parameter Object between layers, we send model object as a parameter.

create function in TransactionService named AddTransaction and pass Transaction as a parameter.

**public** **boolean** AddTransaction (**Transaction** transaction) {

// Create Transaction

}

# Specific Refactorings that you will implement in Milestone 4

**Refactoring 1:** Large Class in the buddi.java class

The buddi class has several responsibilities, to fix that we have to extract methods that have different responsibilities than the class in which it will reduce the size and the complexity of class.

**Refactoring 2:** Fix a complex if/else statements in FilteredLists.java by introducing the state pattern.

**Refactoring 3:** Feature envy between PrefsModel.java class and PrefsModelBean.java class. where more than one method should be moved from the PrefsModel class to PrefsModelBean class as they are interested in the data from PrefsModelBean class

**Refactoring 4:** Large Class in the PrefsModel.java class.

Move the methods that have responsibilities that are different from the class actual responsibility class in which it will reduce the size and the complexity of class.

# References:

1) <http://findbugs.sourceforge.net/index.html>

2) http://buddi.digitalcave.ca/