

## SOEN 6471: Advanced Software Architectures

## Winter 2014

# BUDDI – Personal Budget

# 

# Team Members

Amer Al-Mahdawi (6411754)

Khalid Altoum (6413439)

Gustavo Barbieri Pereira (6273033)

Nikita Parashar (6766994)

Vishnu Vardhan Rajmohan (6688799)

# Contents

[Team Members 1](#_Toc384927693)

[Contents 2](#_Toc384927694)

[Our Repository: 3](#_Toc384927695)

[Project Description 3](#_Toc384927696)

[Ideal Architecture from M2 3](#_Toc384927697)

[Mapping Between Conceptual Classes and Actual Classes 4](#_Toc384927698)

[Class Diagram 4](#_Toc384927699)

[Relationship between classes: 6](#_Toc384927700)

[Discrepancies between concept classes and the actual classes 8](#_Toc384927701)

[Reverse Engineering Tool Used - Yatta “UML Lab” 8](#_Toc384927702)

[Two Classes and Their Relationship 9](#_Toc384927703)

[Code Smells and System Level Refactoring 10](#_Toc384927704)

[Code Smell 1: God Class (or Large Class) 10](#_Toc384927705)

[Code Smell 2: Long Method 13](#_Toc384927706)

[Code Smell 3: Duplicate Code 13](#_Toc384927707)

[Code Smell 4: Switch Statements 13](#_Toc384927708)

[Code Smell 5: Feature Envy 14](#_Toc384927709)

[Code Smell 6: Conditional Complexity 14](#_Toc384927710)

[Specific Refactoring That Will Be Implemented In Milestone 4 16](#_Toc384927711)

[References 19](#_Toc384927712)

Our Repository: <https://github.com/gubp81/BUDDI>

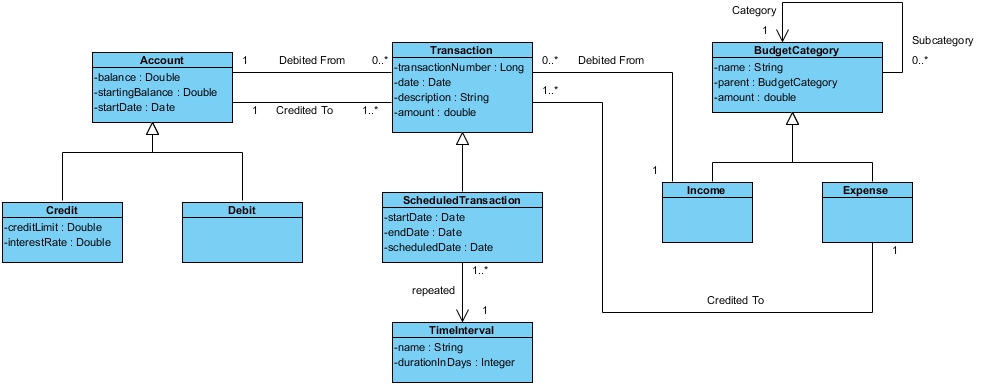
# Project Description

Buddi is a software developed for managing personal finance and budgeting. The goal of this application is to help people with little or no financial background. It can be run on almost any platform (Windows, Mac OS X, Linux, and many other operating system) which has java virtual machine installed (Java 1.6 or higher). Buddi is an open source software developed by *Wyatt Olson*, released under GNU General Public Licenses, that allows anyone to access, modify, and add/remove features to the source code preferably. Its first release (version 0.9 Beta Release) was in May 2006 and its most current stable release is Buddi-3.4.1.11 which was released in August 2013.

Ideal Architecture from M2

Given below is the ideal architecture model that we revised for M2. It shows all the conceptual classes for Buddi Project:

Figure 1. Ideal Architecture



“org.homeunix.thecave.buddi.**model** and org.homeunix.thecave.buddi.**model.impl**” are the packages which defines all the necessary interfaces and classes required for managing application business functionality.

# Mapping Between Conceptual Classes and Actual Classes

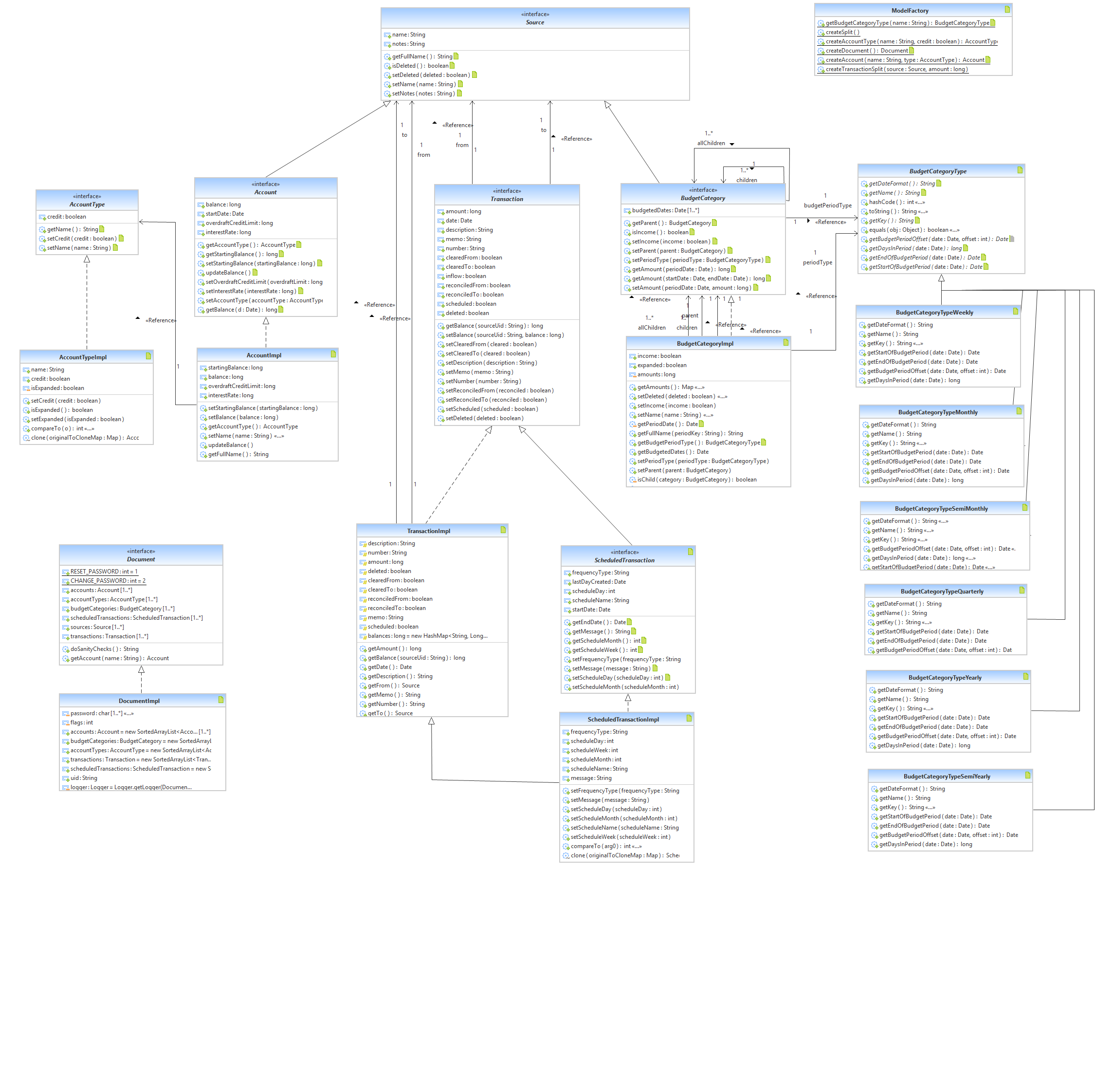
|  |  |  |  |
| --- | --- | --- | --- |
| **CONCEPTUAL CLASSES** | **ACTUAL INTERFACES** | **ACTUAL CLASSES** | **COMMENTS** |
| Account  Debit, Credit | Account | AccountImpl | N/A |
| N/A | AccountType | AccountTypeImpl | It is anti-pattern to OOP polymorphism principle. |
| Transaction | Transaction | TransactionImpl | N/A |
| ScheduledTransaction | ScheduledTransaction | ScheduledTransactionImpl | N/A |
| BudgetCategory  Income, Expense | BudgetCategory | BudgetCategoryImpl | N/A |
| TimeInterval | BudgetCategoryType | BudgetCategoryTypeImpl | Developer applied Strategy Design Pattern on this class for selecting the desired time interval i.e. weekly, monthly etc. |
| - | Source | SourceImpl | The developer decided to use this class as a source and destination of the Transaction. Unfortunately, the source class is not a proper super-class for Account and BudgetCategory classes as this hierarchy does not satisfy IS-A relationship of inheritance. |

# Class Diagram

The Buddi project follows **Open and closed principle** which enables software entities like classes, modules and functions should be open for extension but closed for modifications. Declaring an interface then implementing it is better than starting with developing the concrete class. This offers flexibility to the code by adding new functionality with minimum changes in the existing code.

The class diagram below shows all the actual classes involved in package **model** of project BUDDI. However we have removed some of the classes from the diagram in order to reduce the size of the class diagram and fit into one page but we captured all the fundamental classes necessary to understand the application.

Figure 2.Actual Class Diagram



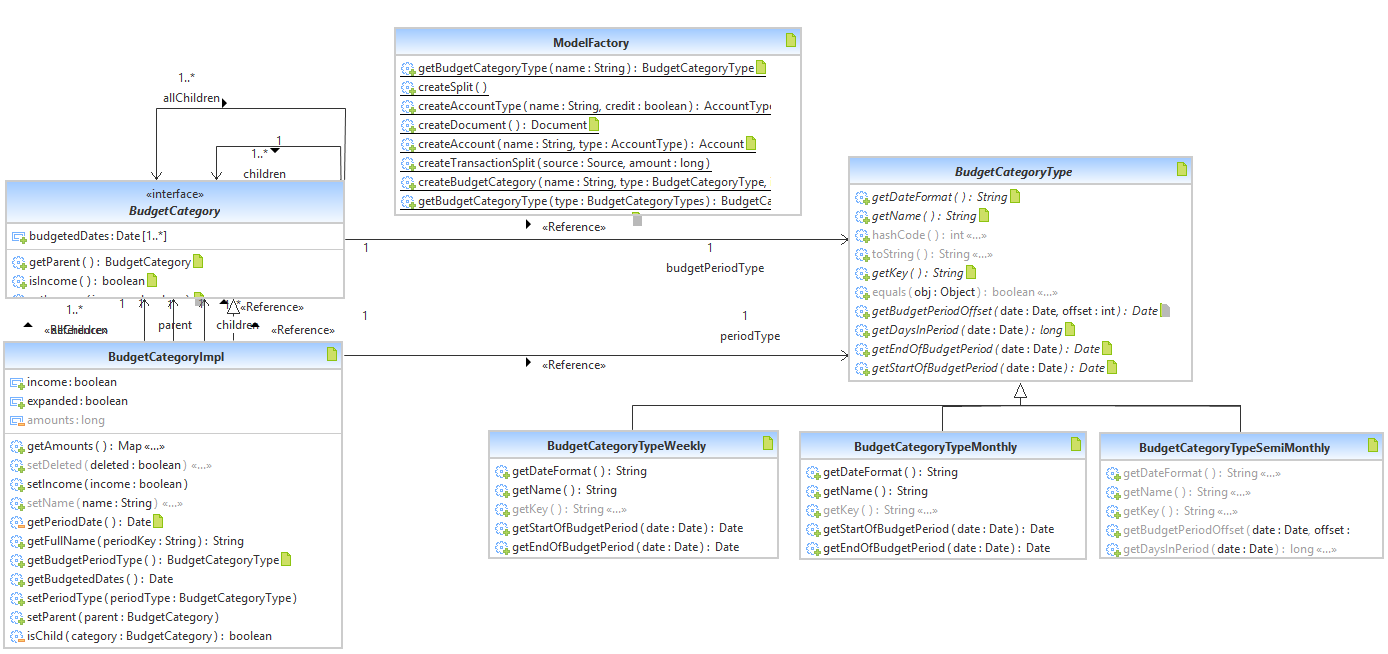
**Different techniques used to build model:**

* Manual class analysis using Eclipse IDE and Net beans.
* Static code analysis using SLOC Counter, JDeodarant and Checkstyle tool.
* Reverse engineering tool to visualize the class hierarchy and dependencies.

# Relationship between classes:

The following description explains relationship among different classes of interest which covers all the fundamental functionalities of the application.

Figure 3.Applying Strategy Pattern along with Factory Pattern

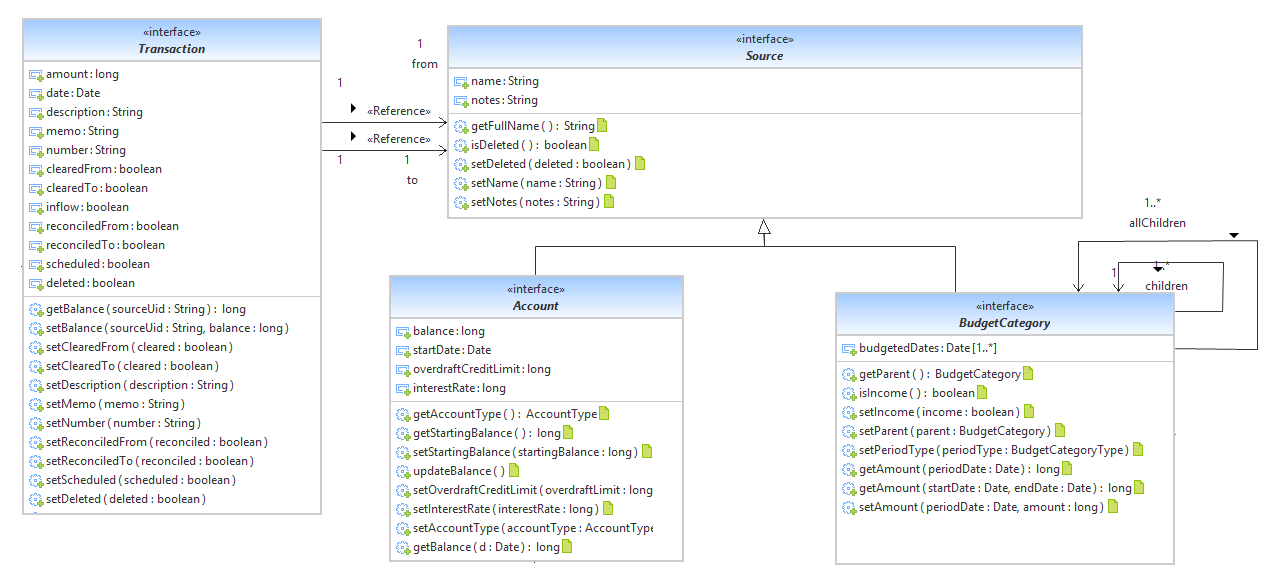


* The above figure shows few classes that follows a combination of Strategy and Factory pattern along with singleton pattern which allows creation of single object.

**BudgetCategoryType** interface acts as a strategy class that will be common to all the supported algorithm which is implemented in number of concrete classes named **BudgetCategoryTypeWeekly, BudgetCategoryTypeMonthly, BudgetCategoryTypeSemiWeekly.** These concrete classes are following the similar kind of algorithm that is nothing but the calculation of different starting and ending duration of budget period based on different time interval. The developer also implemented **ModelFactory** class which helps in creating concrete objects for **BudgetCategoryType** which in turn will be accessed by client class which is **BudgetCategory** class.

* The below figure shows **Source** interface which in turn is extended by Account and BudgetCategory which is not an optimal fabrication because as per the business logic, for inheritance a class must have **Is-A** relationship. Here, **Account** is not a **Source** and **BudgetCategory** Is not a **Source.** But in order to reduce the coupling between **Transaction** & **Account** and **Transaction** & **BudgetCategory** class the developer have introduced **Source** class.

Figure 4. Source Interface and its relationship with Account and BudgetCategory



* **Budget Category** interface also have reflexive association within itself as a category may have sub categories within itself such as entertainment category may have cable, TV or internet as sub category of it, which we already described in domain model description that we submitted in M2. **BudgetCategory** interface is implemented by a single class **BudgetCategoryImpl** but the optimum solution is to implement this interface by 2 sub-classes naming them as **Income** and **Expenses**. **Income class** will add amount to the total balance and **Expense class** will deduct amount from the total balance. Implementation of these two classes from **BudgetCategoryImpl** will make **Transaction** class to directly credit amount to **Expenses** and debit amount to **Income** class.
* The **AccountType** interface is implemented by **AccountTypeImpl** class and **Transaction** interface is implemented by **TransactionImpl** class. **Transaction** class implements updateBalance() method which is updating the current balance of the user into **Account** class which is either credit from the account or debit in to the account.

# Discrepancies between concept classes and the actual classes

* As per the business logic Buddi have **Account** and **AccountType** in actual class diagram which was an Anti-pattern which in domain model is implemented by **Account** abstract class which is extended by two classes named **Debit** and **Credit.**
* Implementation of **Source** class which is extended by **Account** and **BudgetCategory** in the actual class diagram doesn’t follow Is-A relation of inheritance. While in domain model we have **Transaction** class which is in association with **Account** and **BudgetCategory** class replacing **Source**.
* In Domain model we have **TimeInterval** class which is replaced by **BudgetTypeCategory** which is extended by many classes like **BudgetCategoryTypeWeekly**, **BudgetCategoryTypeMonthly, BudgetCategoryTypeMonthly** etc. **BudgetTypeCategory** name is not good as per naming convention but its implementation into different subclasses following Strategy Design Pattern was beneficial for architecture of the system.

# **Reverse Engineering Tool Used** - Yatta “UML Lab”

Reverse engineering tools allows us to dissemble and analyze different components of the program that can either help us for maintaining program or for creating a new program. The reverse engineering tool that we used for our project is Yatta “UML Lab” which is freely available online tool that helps the developers to create adjustment in the source code and diagrams. This automation tool provides flexibility in creation of different concept classes for the class diagram. To visualize models as diagram this tool uses Unified Modeling Language. This tool is able to generate different patterns and diagram for user defined languages that will be helpful for maintenance and care of the software [1].

# Two Classes and Their Relationship

Classes of concern are **AccountImpl** and **TransactionImpl.**

public class **TransactionImpl** extends ModelObjectImpl implements Transaction {

protected long amount;

protected Source from;

protected Source to;

protected final Map<String, Long> balances = new HashMap<String, Long>();

public Source **getFrom**() {

return from;

}

public Source **getTo**() {

return to;

}

public void **setBalance**(String sourceUid, long balance) {

this.balances.put(sourceUid, balance);

}

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

if (this.getTo() != null

&& this.getFrom() != null

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

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

&& (this.getTo() instanceof BudgetCategory

|| this.getFrom() instanceof BudgetCategory {

this.clearedTo = cleared; } }

public void **setClearedTo**(boolean cleared) { ... }

public void **setReconciledFrom**(boolean reconciled) { ... }

public void **setReconciledTo**(boolean reconciled) { }

public class **AccountImpl** extends SourceImpl implements Account {

private long startingBalance;

private long balance;

private Day startDate;

private AccountType type;

public void **updateBalance**(){

long balance = this.getStartingBalance();

List<Transaction> transactions = getDocument().getTransactions(this);

for (Transaction transaction : transactions) {

try {

if (!transaction.isDeleted()){

if (transaction.getTo().equals(this)){

**balance += transaction.getAmount();**

**transaction.setBalance(this.getUid(), balance);}**

else if (transaction.getFrom().equals(this)){

**balance -= transaction.getAmount();**

**transaction.setBalance(this.getUid(), balance);}}**

setBalance(balance);}

public long **getBalance**(Date d) {

if (getDocument() == null)

return 0; //Document not set; not valid. Possibly throw exception?

if (d.before(getStartDate()))

return getStartingBalance();

List<Transaction> ts = getDocument().getTransactions(this, getStartDate(), d);

if (ts.size() > 0){

....

# Code Smells and System Level Refactoring

Buddi source code is composed of 32 packages –as mentioned in the first milestone (M1) document. The main package that includes all the classes responsible for implementing most of the features and computation of the application is:

*src.org.homeunix.thecave.buddi.model.impl*

By inspecting all the 22 classes in this package, we discovered different major and minor code smells.

## Code Smell 1: God Class (or Large Class)

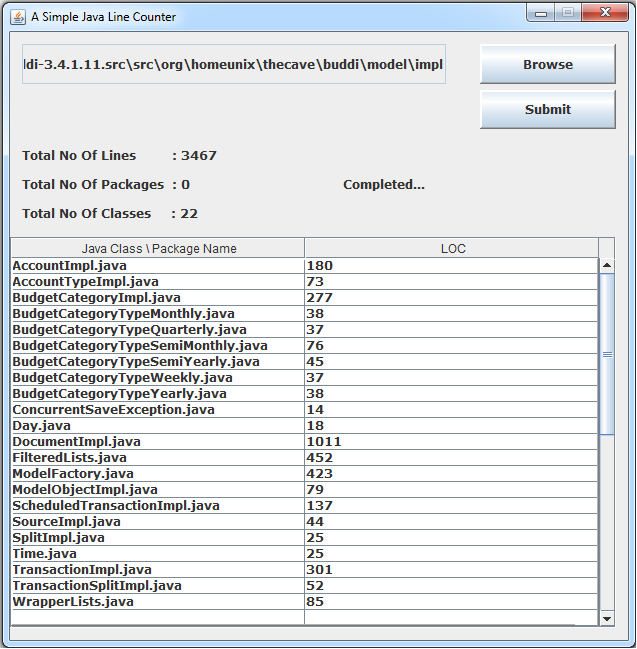


Figure 5

Five classes showed signs of God Class code smell as illustrated in Figure 5 above. They are:

1. AccountImpl:

* handling too much responsibilities: All Balance maintenance (methods *setBalance( )*, *updateBalance( )*, Credit Limit (methods *getOverdraftCreditLimit( )*, *setOverdraftCreditLimit( )* ) and Interest Rate (methods *getInterestRate( )* and *setInterestRate( )* ).
* using many attributes from external classes, directly or via accessor methods: e.g.:

List<Transaction> ts = getDocument().getTransactions(**this**);

1. BudgetCategoryImpl:

* handling too much responsibilities: E.g. Handling Budget Amount and Budgedt Period ( methods *getAmount( ), setAmount( )* ) and Budgedt Period ( methods *overlapStartEndBudgetPeriod( ),getPeriodKey( ), getPeriodType( ), setPeriodKey( ), setPeriodType( )…*).
* excessively large and complex due having methods with high cyclomatic complexity and nesting level, and
* Non-cohesive in terms of how class attributes are used by the internal methods.

1. DocumentImpl

It’s also handling too much responsibilities. Below figure 6.ashows this class before refactoring (only the attributes and methods relevant are shown) and figure 6.b shows the UML of the target classes after refactoring.



Figure 6.a

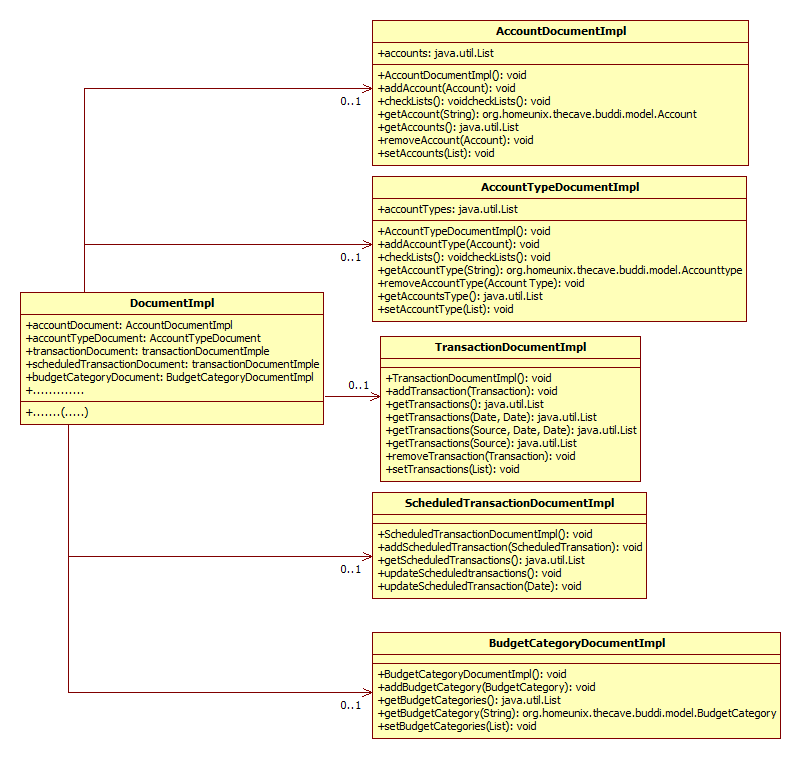


Figure 6.b

1. FilteredLists:

* This class contains many static classes that should be extracted into separated java classes. It registers the data model for change events, and updates the filtered list accordingly and also returns a list of all transactions which are associated with a given source. BuddiFilteredList<T>.
  + Class TransactionListFilteredBySource returns a list of all transaction.
  + Class TransactionListFilteredByDate returns a list of all transactions which fall between startDate and endDate.
  + Class AccountListFilteredByType extends BuddiFilteredList<Account> Returns a list of all the Accounts associated with the given type
  + Class AccountListFilteredByDeleted extends BuddiFilteredList<Account> Returns a list of all accounts included in the constructor, with deleted ones removed if the Preferences state that you should do it.
* Excessively large and complex due having methods with high cyclomatic complexity and nesting. One of this methods is acceptDate( ).

1. TransactionImpl:

* handling too much responsibilities: E.g., Handling account movements ( methods getTo( ), getFrom( ), setTo( ), setFrom( )*…*) Handling Amount and Balance( methods *getAmount( ), setAmount( ), getBalance( ), setBalance( )* ).
* Excessively large and complex due having methods with high cyclomatic complexity, as methods setReconciledFrom( )/setReconciledTo( ) and setClearedFrom and setClearedto.

## Code Smell 2: Long Method

Many methods were too complex and too long to be understood and doing too many computations. And many times a single method is doing totally different things. E.g. Method checkValid(...) in class “DocumentImpl

## Code Smell 3: Duplicate Code

Some duplicated code was found in different methods. The simplest duplicated code problem is with methods from same class. E.g. methods setReconciledFrom and setReconciledTo of class TransactionImpl. They are using same conditions on a long IF statement.

public void setReconciledFrom(boolean reconciled){

if (this.getTo() != null && this.getFrom() != null

….

}

public void setReconciledTo(boolean reconciled){

if (this.getTo() != null && this.getFrom() != null

….

}

public void setReconcil ()

{

if (this.getTo() != null && this.getFrom() != null

….

}

## Code Smell 4: Switch Statements

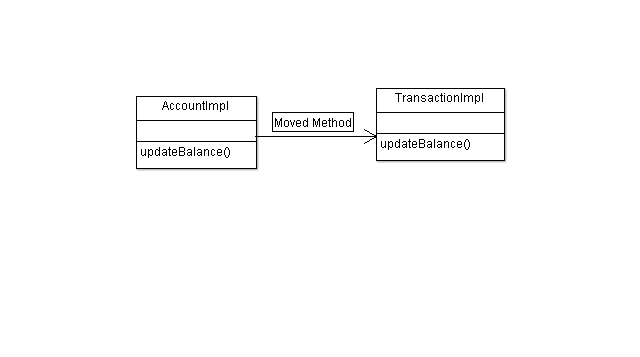
We found some methods with if-elseif-else statement with twelve conditional branches that works exactly the same way any switch statement do. It is a good candidate to apply “Replace Type Code with State/Strategy” refactoring. One of this methods is:

acceptDate(...) and it exists in class “FilteredLists.TransactionListFilteredBySearch”.

This method has a long if-else statement with 10 conditions.

## Code Smell 5: Feature Envy

We found some methods that seems more interested in another class other than the one it actually is in. One of this methods is updateBalance() declared in class AccountImpl. The only reference for this method is in class TransactionImpl.



## Code Smell 6: Conditional Complexity

Many methods have conditions too complex and too long to be understood. Some conditional logic controls which of several variants of a calculation to execute. One of this methods is getAmount(Date periodDate) from class BudgetCategoryImpl.

**public long getAmount(startDate , endDate )**

**private long sameStartDateBudgetPeriod**

**(startDate , endDate)**

**private long overlapStartEndBudgetPeriod**

**(startDate , endDate )**

INTRODUCING EXPLANATORY VARIABLES

**final Date startDateOfBudgetPeriod = getBudgetPeriodType().**

**getStartOfBudgetPeriod(startDate);**

**final Date endDateOfBudgetPeriod = getBudgetPeriodType().**

**getStartOfBudgetPeriod(endDate);**

**final Date startBudgetPeriodOffset = getBudgetPeriodType().**

**getBudgetPeriodOffset(startDate, 1);**

# Specific Refactoring That Will Be Implemented In Milestone 4

We chose two classes to apply the refactoring techniques required to remove the code smells mentioned in the previous section; class “DocumentImpl” and class “FilteredLists.TransactionListFilteredBySearch”. The details are followed:

## Fixing the Long Method Code Smell:

Method checkValid(...) in class “DocumentImpl” (which has 100 line of code including comments and spaces) has been chosen to remove this particular smell from it. To do so, we will apply the following steps:

* Extract Method: in order to divide the long method into smaller and simpler methods, and give them appropriate names.
* Move Method: if needed.
* Compilation and testing is applied after each move.

## Fixing Duplicated Code Smell:

The methods setReconciledFrom and setReconciledTo of class TransactionImpl are using same conditions on a long IF statement. Same thing is happing with methods setClearedFrom and setClearedto of same class. To remove these code smell we will apply the following steps:

* The duplicated code will be moved to a new method.
* Replace the duplicated code by the new method.
* Compilation and testing is applied after each move.
* Repeat those steps for each duplicated code found

## Fixing the Switch Statement Code Smell:

For method “acceptDate(...)” in class “FilteredLists.TransactionListFilteredBySearch” we will apply:

* Replace Type Code with State/Strategy. Then
* Replace Conditional with Polymorphism.
* Compilation and testing is applied after each move.

Below is the code of the target method:

**public** **class** FilteredLists

{

**public** **static** **class** TransactionListFilteredBySearch **extends** BuddiFilteredList<Transaction> {

**private** **boolean** acceptDate(Transaction t) {

**if** (**null** == dateFilter || TransactionDateFilterKeys.*TRANSACTION\_FILTER\_ALL\_DATES* == dateFilter)

{

**return** **true**;

}

Date today = **new** Date();

**if** (TransactionDateFilterKeys.*TRANSACTION\_FILTER\_TODAY* == dateFilter) {

**return** DateUtil.*isSameDay*(today, t.getDate());

}

**else** **if** (TransactionDateFilterKeys.*TRANSACTION\_FILTER\_YESTERDAY* == dateFilter) {

**return** DateUtil.*isSameDay*(DateUtil.*addDays*(today, -1), t.getDate());

}

**else** **if** (TransactionDateFilterKeys.*TRANSACTION\_FILTER\_THIS\_WEEK* == dateFilter) {

**return** DateUtil.*isSameWeek*(today, t.getDate());

}

**else** **if** (TransactionDateFilterKeys.*TRANSACTION\_FILTER\_THIS\_SEMI\_MONTH* == dateFilter) {

BudgetCategoryType semiMonth = **new** BudgetCategoryTypeSemiMonthly();

**return** (semiMonth.getStartOfBudgetPeriod(**new**

Date()).equals(semiMonth.getStartOfBudgetPeriod(t.getDate())));

}

**else** **if** (TransactionDateFilterKeys.*TRANSACTION\_FILTER\_LAST\_SEMI\_MONTH* == dateFilter) {

BudgetCategoryType semiMonth = **new** BudgetCategoryTypeSemiMonthly();

**return** (semiMonth.getStartOfBudgetPeriod(semiMonth.getBudgetPeriodOffset(**new** Date(),

1)).equals(semiMonth.getStartOfBudgetPeriod(t.getDate())));

}

**else** **if** (TransactionDateFilterKeys.*TRANSACTION\_FILTER\_THIS\_MONTH* == dateFilter) {

**return** DateUtil.*isSameMonth*(today, t.getDate());

}

**else** **if** (TransactionDateFilterKeys.*TRANSACTION\_FILTER\_LAST\_MONTH* == dateFilter) {

**return** DateUtil.*isSameMonth*(DateUtil.*addMonths*(today, -1), t.getDate());

}

**else** **if** (TransactionDateFilterKeys.*TRANSACTION\_FILTER\_THIS\_QUARTER* == dateFilter) {

**return** DateUtil.*getStartOfDay*(DateUtil.*getStartOfQuarter*(today)).before(t.getDate());

}

**else** **if** (TransactionDateFilterKeys.*TRANSACTION\_FILTER\_LAST\_QUARTER* == dateFilter) {

**return** DateUtil.*isSameDay*(DateUtil.*getStartOfQuarter*(DateUtil.*addQuarters*(today, -1)),

DateUtil.*getStartOfQuarter*(t.getDate()));

}

**else** **if** (TransactionDateFilterKeys.*TRANSACTION\_FILTER\_THIS\_YEAR* == dateFilter) {

**return** DateUtil.*isSameYear*(today, t.getDate());

}

**else** **if** (TransactionDateFilterKeys.*TRANSACTION\_FILTER\_LAST\_YEAR* == dateFilter) {

**return** DateUtil.*isSameYear*(DateUtil.*addYears*(today, -1), t.getDate());

}

**else** {

Logger.*getLogger*(**this**.getClass().getName()).warning("Unknown filter pulldown: " + dateFilter);

**return** **false**;

}

}

}

}

## Fixing the Feature Envy Code Smell:

Method updateBalance() declared in class AccountImpl has been chosen to be moved to class TransactionImpl. To do so, we will apply the following steps:

* Extract a new method named updateTheBalance() from updateBalance() declared in AccountImpl class.
* Move updateTheBalance() method from AccountImpl to TransactionImpl.
* Extract 4 new methods from updateBalance() and name them as updateDestinationAccountBalance(), updateSourceAccountBalance(), updateBalanceFromDestinationAccount(),updateBalanceFromSourceAccount().
* Compilation and testing is applied after each move.

**Public void updateBalance()**

**updateDestinationAccountBalance()**

**updateSourceAccountBalance()**

**updateBalanceFromDestinationAccount()**

**updateBalanceFromSourceAccount()**

## Fixing the Conditional Complexity Code Smell:

Method getAmount(Date periodDate) from class BudgetCategoryImpl. has been chosen to be changed to remove code smell Conditional Complexity. To do so, we will apply the following steps:

* Created two new methods from getAmount(Date periodDate) and name them as sameStartDateBudgetPeriod() and overlapStartDateBudgetPeriod().
* Introduce 3 Explaining Variables to public long getAmount() to simplify the complexity of the conditional expression:

final Date startDateOfBudgetPeriod = getBudgetPeriodType().getStartOfBudgetPeriod(startDate);

final Date endDateOfBudgetPeriod = getBudgetPeriodType().getStartOfBudgetPeriod(endDate);

final Date startBudgetPeriodOffset = getBudgetPeriodType().getBudgetPeriodOffset(startDate, 1);

* Compilation and testing is applied after each move.

# References

1. http://www.uml-lab.com/en/uml-lab/