



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
Our Repository:.....	3
Project Description.....	3
Ideal Architecture from M2:	3
Mapping Between Conceptual Classes and Actual Classes.....	3
Class Diagram.....	4
Relationship and Discrepancies between Classes	5
Reverse Engineering Tool Used	7
Two Classes and Their Relationship.....	7
Code Smells and System Level Refactoring.....	9
Specific Refactoring That Will Be Implemented In Milestone 4.....	10
References	13

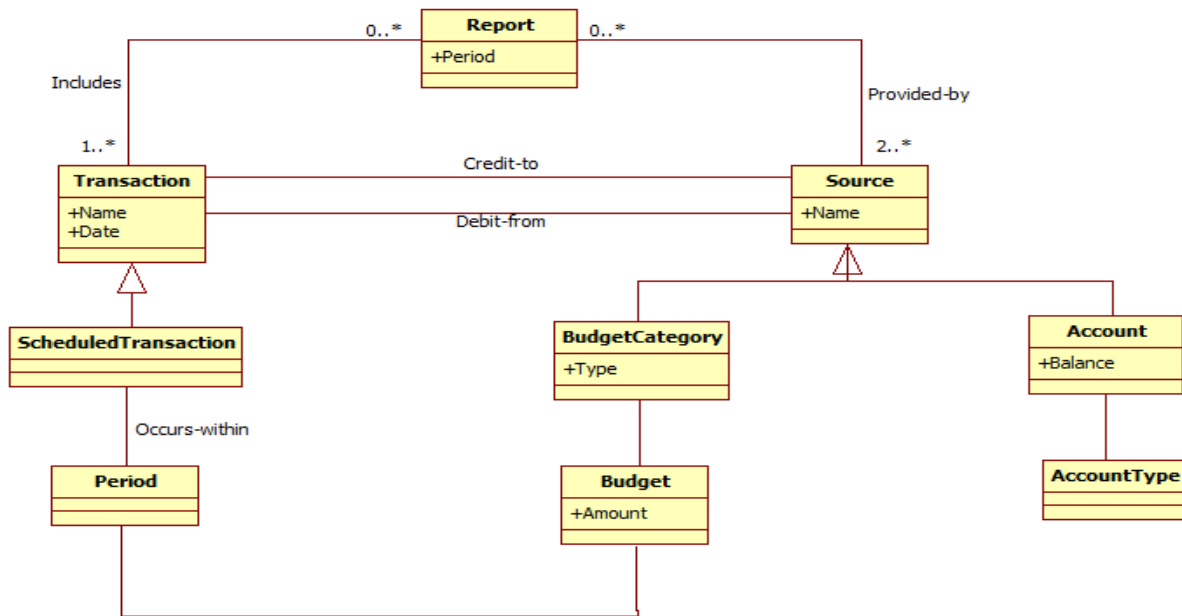
Our Repository: <https://github.com/gubp81/BUDDI>

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 submitted for M2. It shows all the actual 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
Report	Document	DocumentImpl	DocumentImpl was a large class with lots of responsibilities.

Transaction	Transaction	TransactionImpl	N/A
Source	Source		Usage is not optimal for fabrication, but its utilization reduces the complexity of the code.
Scheduled Transaction	Scheduled Transaction	ScheduledTransactionImpl	N/A
BudgetCategory	BudgetCategory	BudgetCategoryImpl	
Budget			
Period	BudgetCategoryType	BudgetCategoryType(Weekly, Monthly, Semi-monthly)	Due to multiple duration, the developer applied the strategy pattern along with factory pattern to generate multiple subclasses in order to maintain weekly, monthly, quarterly etc.
Account	Account	AccountImpl	N/A
Account Type	AccountType	AccountTypeImpl	N/A

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.

[illegible]

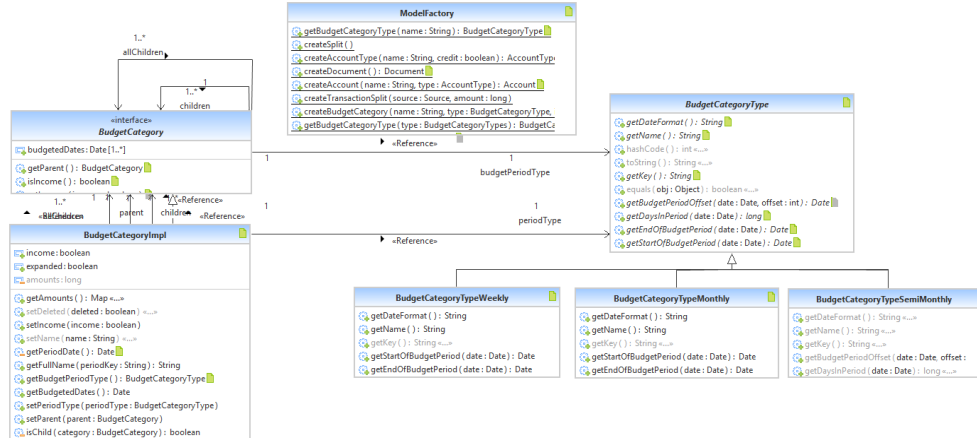
- Manual class analysis using Eclipse IDE.
- Static code analysis using SLOC Counter, JDeodrant and Checkstyle tool.
- Reverse engineering tool to visualize the class hierarchy and dependencies.

Communication among different packages occurs with the help of different interfaces.

Criteria 1: It exists in the Actual Class Diagram but we missed it.

Criteria 3: Missed by the Actual Class Diagram but mentioned by us.

Figure 3. Applying Strategy Pattern along with Factory Pattern

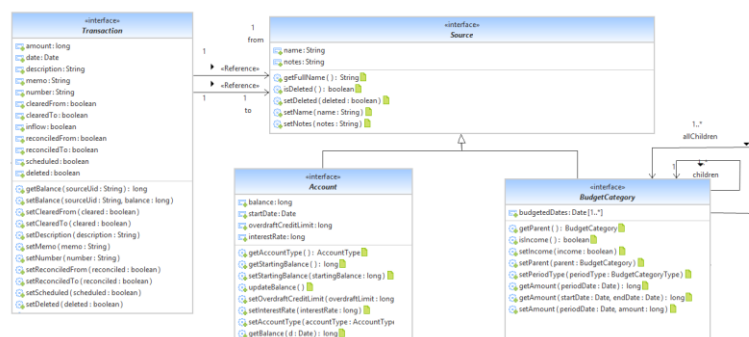


- The above figure shows few entities that follows a combination of Strategy and Factory pattern along with singleton pattern which allows creation of single object.

Budget Category Type interface which is implemented based on period duration that we defined in our domain model i.e. based on weekly, monthly and semimonthly type basis. The developer used strategy pattern along with factory pattern to be able to implement Budget Category Type interface with different concrete classes applying different duration of time.

Budget Category interface also have reflexive association within itself as a category may have sub categories such as entertainment category may have cableTV as sub category of it, which we already described in domain model description that we submitted in M2 but it does not exist in the domain model diagram. Budget Category interface is implemented by a single class **BudgetCategoryImpl** but the optimum solution is to implement this interface by 2 sub-classes named Income and Expenses. **Income class** will add amount to the total balance and **Expense class** will deduct amount from the total balance. This discrepancy follows Criteria 3.

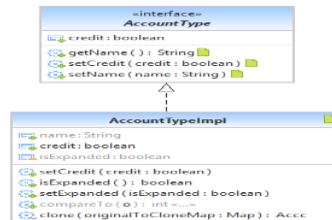
Figure 4. Source Interface and its relationship with Account and BudgetCategory



- The above figure shows **Source** interface is implemented by a class called **SourceImpl** which is not an optimal fabrication but used to reduce the coupling between Transaction & Account and Transaction & Budget Category.

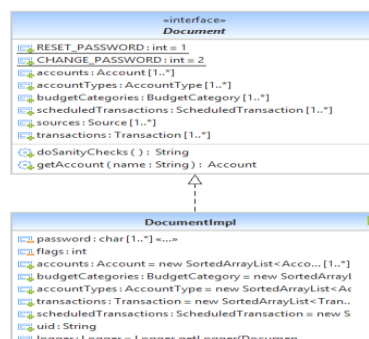
- The below figure shows the AccountType interface which is implemented by AccountTypeImpl subclass. However, the optimal solution is to implement this interface by **Debit and Credit class**. This type of discrepancy follow Criteria 2.

Figure 5.AccountType Implementation



- Document** interface implemented by **DocumentImpl** class which is a god class coupled to many other classes. We used **Report** as synonym for this class and it should be implemented by many classes having high cohesion. To simplify the actual classes, we removed all the non-essential classes which in turn increased the readability of the diagram.

Figure 6.Document Implementation



Reverse Engineering Tool Used - Yatta “UML Lab”

Reverse engineering tools allows us to disassemble 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 **Account**.

Each Account has an AccountType which affects its behaviour. If it's a Credit the amount will be deducted from the Account which the client have to pay back and if it's a debit, amount will be added to the Account.

```
package org.homeunix.thecave.buddi.model.impl;

public class AccountImpl extends SourceImpl implements Account {

    private long startingBalance;

    private long balance;

    private long interestRate;

    private Day startDate;

    private AccountType type;

    public void updateBalance(){ }

    public long getInterestRate() { }

    public void setInterestRate(long interestRate) throws InvalidValueException { }

}
```

```
package org.homeunix.thecave.buddi.model;

public interface Account extends Source{

    public AccountType getAccountType();

    public long getBalance();

    public long getBalance(Date d);

    public Date getStartDate();

    public void setStartDate(Date startDate);

    public long getStartingBalance();

    public void updateBalance();

    public long getOverdraftCreditLimit();

    public long getInterestRate();

}
```


Code Smells and System Level Refactoring

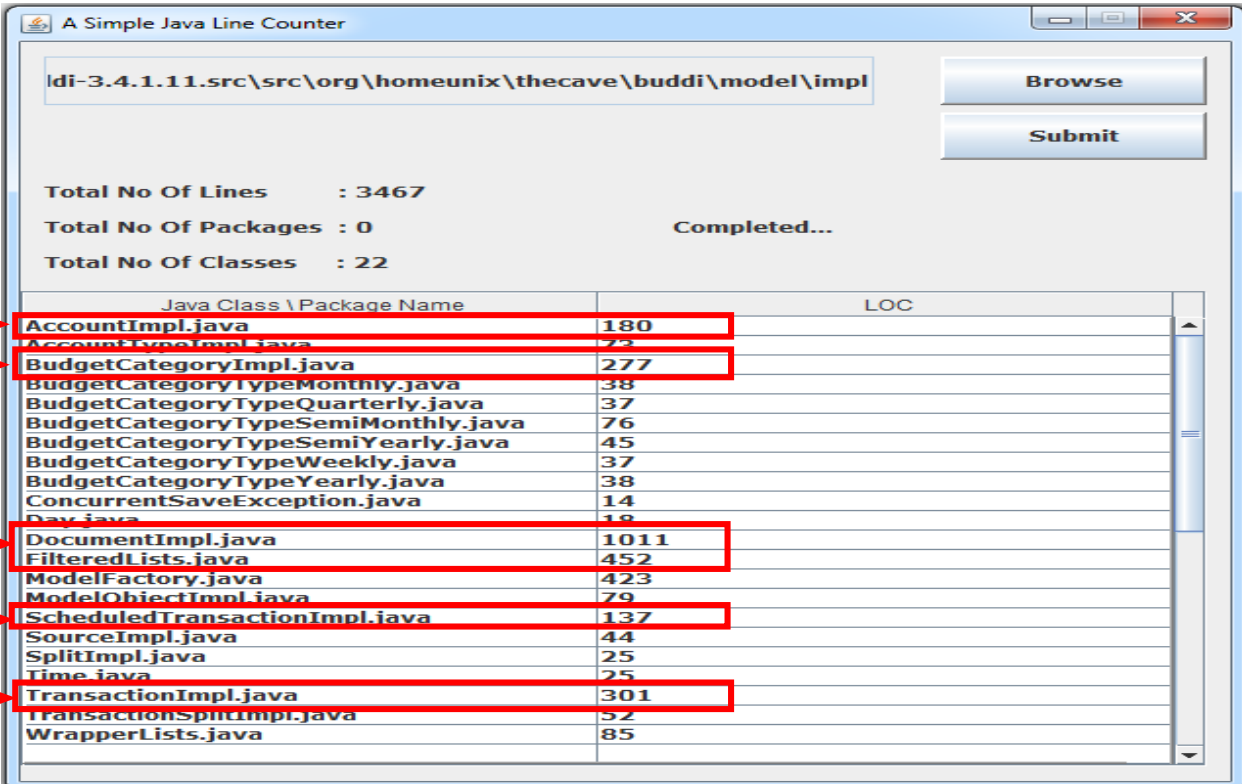
Buddi source code is composed of 32 packages –as mentioned in the first milestone (M1) document. The main package that includes all the classes responsible for implementing most of the features and computation of the application is:

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

By inspecting all the 22 classes in this package, we discovered different major and minor code smells. The major ones are described below:

- God Class: Six classes showed signs of God Class code smell as illustrated in Figure X. They:
 - handling too much responsibilities,
 - use many attributes from external classes, directly or via accessor methods,
 - are excessively large and complex due to having methods with high cyclomatic complexity and nesting level, and
 - Non-cohesive in terms of how class attributes are used by the internal methods.
- Long Method: Many methods were too complex and too long to be understood and doing too many computations.
- Switch Statements: We found a method 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. The name of this method is:

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



A Simple Java Line Counter

Idi-3.4.1.11.src\src\org\homeunix\thecave\buddi\model\impl

Browse

Submit

Total No Of Lines : 3467

Total No Of Packages : 0

Total No Of Classes : 22

Completed...

Java Class \ Package Name	LOC
AccountImpl.java	180
AccountTypeImpl.java	73
BudgetCategoryImpl.java	277
BudgetCategoryTypeMonthly.java	38
BudgetCategoryTypeQuarterly.java	37
BudgetCategoryTypeSemiMonthly.java	76
BudgetCategoryTypeSemiYearly.java	45
BudgetCategoryTypeWeekly.java	37
BudgetCategoryTypeYearly.java	38
ConcurrentSaveException.java	14
Day.java	19
DocumentImpl.java	1011
FilteredLists.java	452
ModelFactory.java	423
ModelObjectImpl.java	79
ScheduledTransactionImpl.java	137
SourceImpl.java	44
SplitImpl.java	25
Time.java	25
TransactionImpl.java	301
TransactionSplitImpl.java	52
WrapperLists.java	85

Figure 2

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 God Class Code Smell:

Different refactoring methods will be applied on class “DocumentImpl” which handles the responsibility of managing five components, namely (Account, AccountType, BudgetCategory, Transaction, and ScheduledTransaction). We assume extracting a separate class for each component. To do so, we will apply the following steps:

1. Extract Class: for each component, and give it an appropriate name.
2. Make a link from the old class to each created class by creating an object of each new class in the old one.
3. Move Field: all the fields related to each component will be moved to the relevant created class.
4. Extract Method: Four methods will be extracted from method “checklists()” -which exists in line 838-.
5. Move Method: all the methods related to each component will be moved to the relevant created class, delegation will be used if needed.
6. Other minor refactoring may applied in order to maintain the design goals like (Rename Method, Hide Method, and Replace Parameter with Method.
7. Compilation and testing is applied after each move.
8. Below figure 9.a shows this class before refactoring (only the attributes and methods relevant are shown) and figure 9.b shows the UML of the target classes after refactoring.

DocumentImpl
+accounts: java.util.List +accountTypes: java.util.List +budgetCategories: java.util.List +scheduledTransaction: java.util.List +transactions: java.util.List +.....
+addAccount(Account): void +addAccountType(AccountType): void +addBudgetCategory(BudgetCategory): void +addScheduledTransaction(ScheduledTransaction): void +addTransaction(Transaction): void -checkLists(): void +getAccount(String): org.homeunix.thecave.buddi.model.Account +getAccounts(): java.util.List +getAccountType(String): org.homeunix.thecave.buddi.model.AccountType +getAccountTypes(): java.util.List +getBudgetCategories(): java.util.List +getBudgetCategory(String): org.homeunix.thecave.buddi.model.BudgetCategory +getScheduledTransactions(): java.util.List +getTransactions(): java.util.List +getTransactions(Date, Date): java.util.List +getTransactions(Source, Date, Date): java.util.List +getTransactions(Source): java.util.List +removeAccount(Account): void +removeAccountType(AccountType): void +removeBudgetCategory(BudgetCategory): void +removeScheduledTransaction(ScheduledTransaction): void +removeTransaction(Transaction): void +setAccounts(List): void +setAccountTypes(List): void +setBudgetCategories(List): void +setScheduledTransaction(List): void +setTransactions(List): void +updateScheduledtransactions(): void +updateScheduledTransaction(Date): void +.....(.....)

Figure 9.a

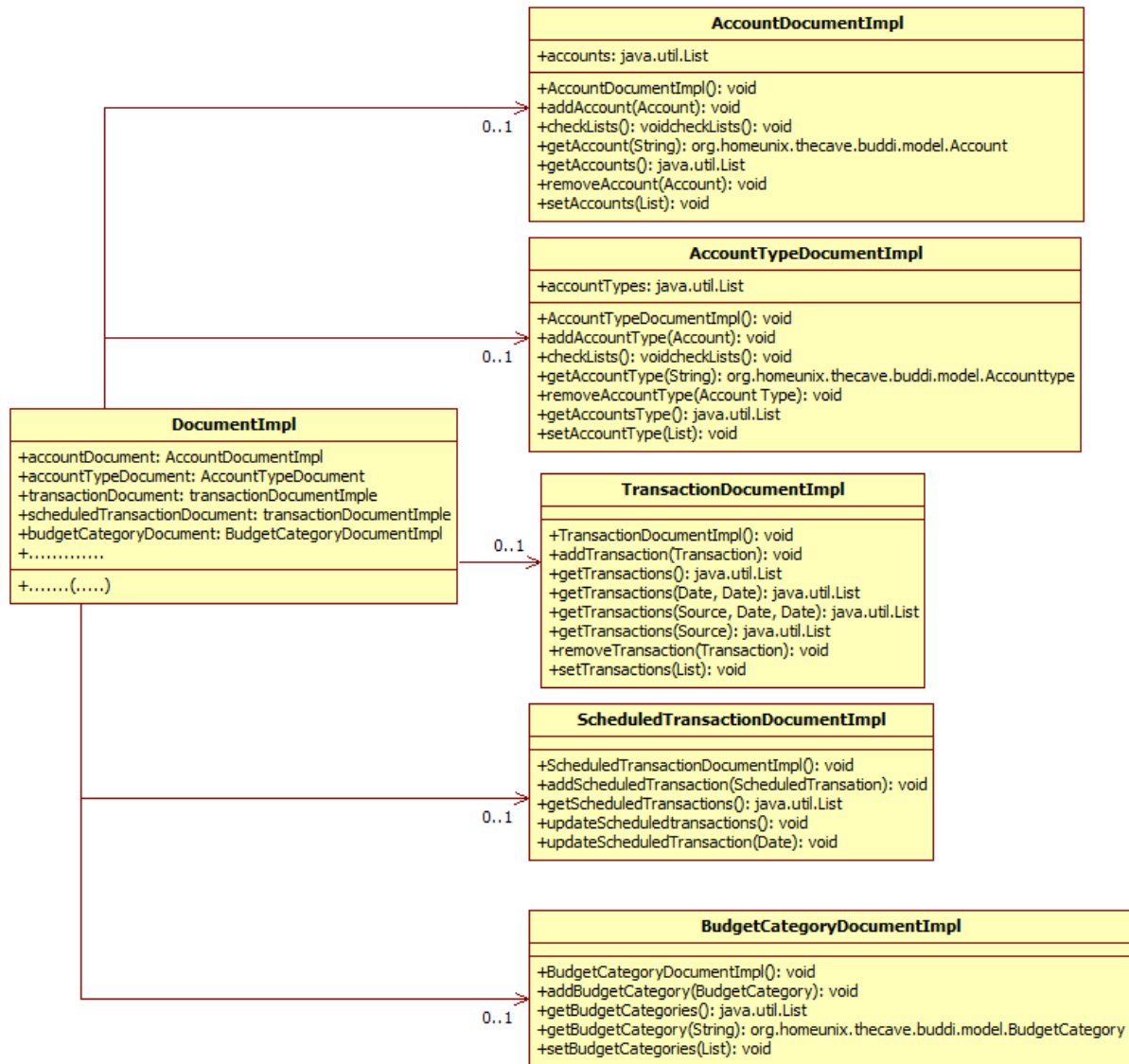


Figure 9.b

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:

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

Fixing the Switch Statement Code Smell:

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

1. Replace Type Code with State/Strategy. Then
2. Replace Conditional with Polymorphism.
3. 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;
}
}
}
}

```

References

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