SOEN 6471 Milestone 3

iFreeBudget System

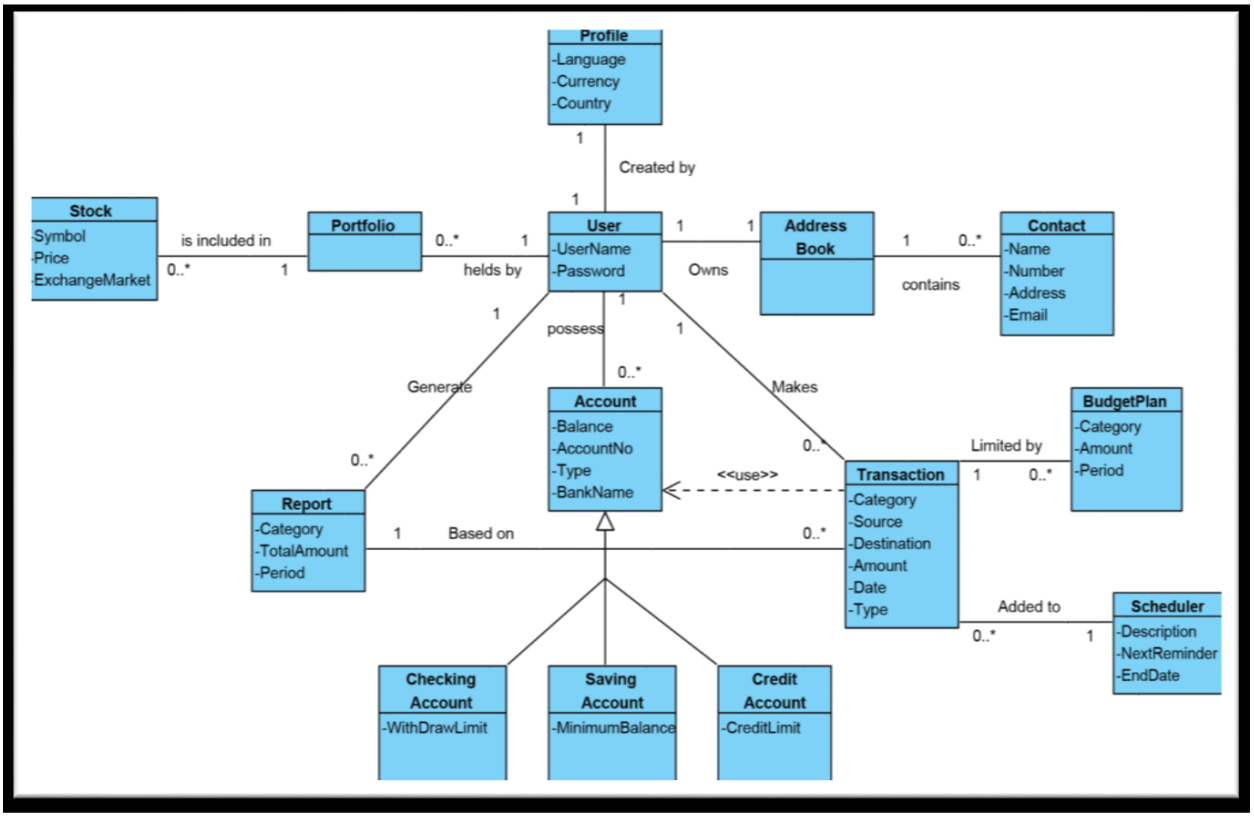
Student Name:	Student Id:
Kishan Shah	06819230
Tirth Patel	06893783
Vivian Michael Gerard	06764061
Zhang Meng	06401368
Ruixing Tan	06783783
Date: March 3rd 2014.	

1. Summary of Project

iFreeBudget is a free, open source expense and budget tracking application for PC and Android platforms. This application is used for financial purpose it has very good feature to manage income, expenses and savings of the company or family. This system also includes features like alarming; user can create reminders for transactions before-hand so that the user can pay his bills within the deadline. Payment of various bills using this application is very intuitive and efficient. One of the best feature of this system is user can generate report of all transaction he had made in particular time. The user is able to analyze his earnings/income and his cash flow details very easily using the graphically represented of reports. System has features like Personal Finance, Budgeting, Scheduling transactions, graphical reporting and Portfolio Management.

2. Class Diagram of Actual System

2.1. Conceptual Class Diagram



2.2 Actual Class Diagram

2.3 Class Diagram Elaboration

Although the actual architecture of the whole iFreeBudget system is very complicated than the domain model which we had imagined in milestone 2, the relationship between the important entities we generated from the source code are similar to the entities (conceptual classes) in milestone 2. Note that, the class diagram is simplified so that only the relevant classes (entities) are shown for analysis except for the scheduler module which we can discuss in detail in later sections.

The actual iFreeBudget system has two major modules **fm**(Financial Management) and **Scheduler.** Some of the commonly used financial terms appear in both domain model and the actual class diagram i.e. **Account**, **Transaction**, **Portfolio**, **Budget** and basic entities such as **User** and **Profile**. Take Account as an example, it has attributes such as id, type, account number, date etc. As we have assumed in the domain diagram and actions such as **AddAccountAction**, **DeleteAccountAction**, **ExportAccountAction** and **GetAccountListAction**.

There are different naming conventions used for the essentially same entities. The **PortfolioEntry** is the same as the **Stock**s, both are parts of the **Portfolio** in their receptive diagrams.

NetWorthHistory is functionally similar to **Report** in the domain diagram. In milestone 2, we have considered Report to include the transactions in a certain period of time which can be shown to the users. In the actual system, the user interface will invoke the **NetWorthHistory** to show the combination of assets and liabilities between a certain periods of time. There are also other types of Report which are directly done by a single operation such as **GetEarningReportAction**, **GetExpReportAction** etc, the calculations are done inside the classes.

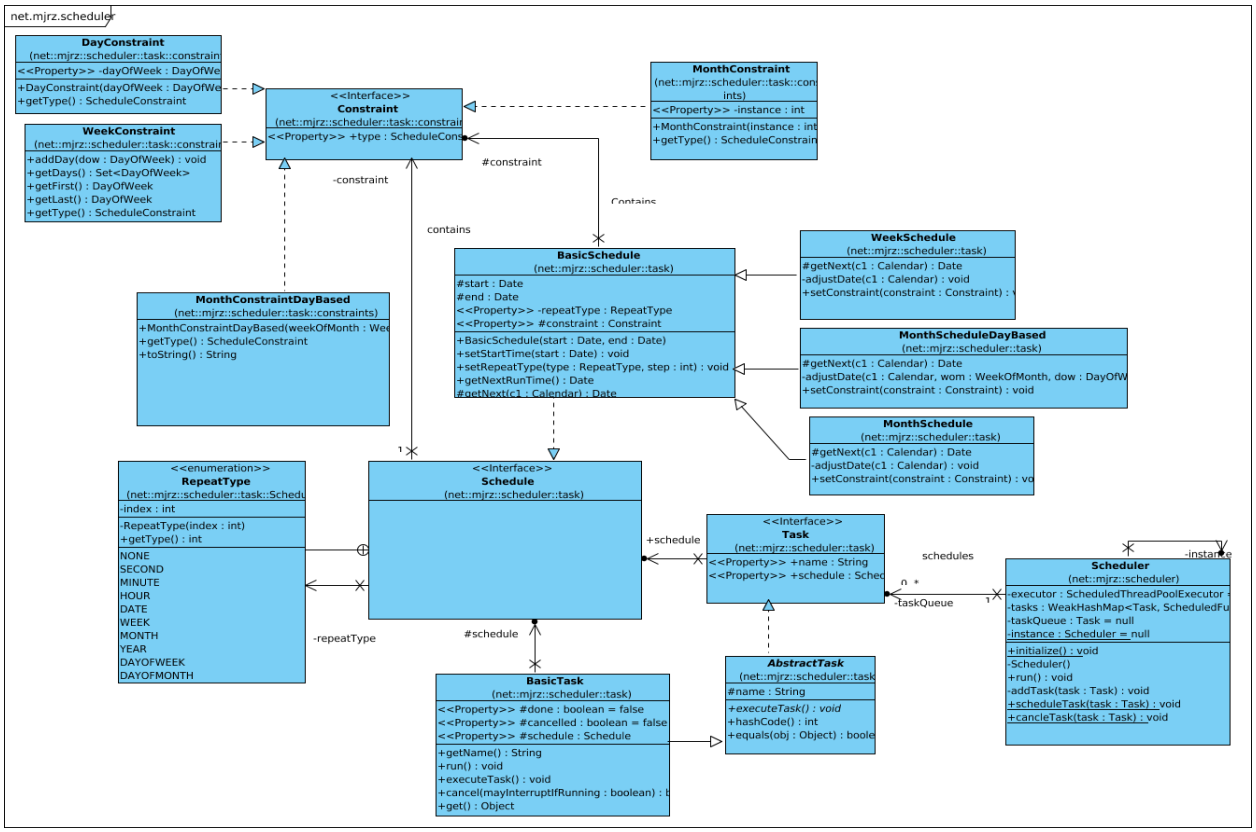
Meanwhile, some differences are also noticeable in the actual class diagram we generated. First, some of the entities turn out to be just attributes in the actual classes while some attributes we imagined become actual classes. In our domain diagram, several different accounts including **ChequeAccount**, **SavingAccount** and **CreditAccount** inherited from the **Account** class to distinguish different categories of the accounts. In the actual class diagram, however, **AccountCatogory** along with **AccountAddress** are the attributes of the **Account** class, different types of the account are defined in the **AccountCatogory**. On the contrary, there is no **AddressBook** class; instead the **Contact**s are listed directly when invoked in the user interface. There is a new class **FutureTransaction**, this is for the transactions to be added in the future, it can also have attribute so that it can be periodical, when the FutureTransactoin need to be added to the **User**, a new instance of Transaction will be created and added to the User instead.

Regarding the architecture of the system, these are the major parts of the system: the entities, the actions, database and user interface. The entities package contains all the entities

we have discussed in milestone 2 and above. Besides, there are entity managers which are responsible for the interaction between the operations on the entities and the database. These operations are contained in the actions package, for instance, **AddAccountAction**, **GetTransactoinAction**, **RemovePortfolioAction** etc. Actions package are very specific for only one certain operation which means this part of the program has low coupling. This is a good programming practice. Above mentioned diagram shows the whole architecture of the financial management system. **FManEntityManager** contains all the methods regarding to the actions, which will perform these action directly with the database.

The reverse engineering tool we use is **Visual Paradigm for UML Standard Edition**. Visual Paradigm for UML (VP-UML) is a UML CASE Tool supporting UML 2, SysML and Business Process Modeling Notation (BPMN), MS Visio 2013 from the Object Management Group (OMG). It can reverse engineer diagrams from code, and provide round-trip engineering for various programming languages.

2.4 Relationship between classes with a sample code



(Class Diagram of Scheduler Module)

The **Scheduler** executes tasks submitted to it for future execution in a background thread. Tasks submitted for execution can be cancelled any time before execution or during execution. The basic workflow of the scheduler is to:

- Initialize the scheduler so that it can run in the background as a separate thread.
- Create -a new Task.
- Create a schedule for the task by setting the start and end time.
- Set recurrence-The RepeatType of the Scheduler determines whether the recurrence should happen monthly/weekly or daily.
- Submit the task.

We consider the following classes of the **Scheduler** module to illustrate the relationship between them:

- **BasicSchedule**
- **MonthSchedule**
- **MonthConstraint**

The MonthSchedule **IS-A** type of BasicSchedule and the MonthSchedule **HAS-A** MonthConstraint.

The **Schedule interface** describes the behaviors like

- void setStartTime(Date date)/setEndTime(Date date)-- start/end time of the task to be scheduled.
- void setRepeatType(RepeatType type, int step)--Sets the type of scheduling that is to be done, whether it is weekly or monthly or daily
- void setConstraint(Constraint constraint)--Sets the constraints for this schedule. Constraints define the dates on which a task will be executed. For example, WeekConstraint can be setup so that the task executes on specified days of the week.
- Constraint getConstraint()--Returns the constraints defined for this schedule

The **BasicSchedule** class implements the **Schedule** interface and defines all the behaviours that were described in the interface. This class constructs a schedule with the specified start and end dates. Also, for getting the specific schedules (such as monthly, weekly etc.) the class should override the **getNext()** method to implement the appropriate next run-time calculation.

The **MonthSchedule** class inherits the parent class **BasicSchedule** and generates a schedule with monthly recurring tasks constrained with **MonthConstraint**.

The **MonthConstraint** class implements a common interface **Constraint**. Constraint can be added to a month's schedule. A constraint defined on a schedule can be used to further tweak the

recurrence behavior. The MonthConstraint class defines a constraint such that, the task will be executed on the specified date of the every month.

The methods that are relevant to the three classes are extracted and shown in the blow snippets:

BasicSchedule

```
public class BasicSchedule implements Schedule, Serializable {

	private RepeatType repeatType;
	protected Constraint constraint;

	protected Date getNext(Calendar c1) {

		if (repeatType == RepeatType.MONTH) {
			c1.add(Calendar.MONTH, step);
			return c1.getTime();
		}
		if (repeatType == RepeatType.YEAR) {
			c1.add(Calendar.YEAR, step);
			return c1.getTime();
		}
		return null;
	}

	@Override
	public void setConstraint(Constraint constraint) {
		this.constraint = constraint;
	}

	@Override
	public Constraint getConstraint() {
		return this.constraint;
	}

}
```

MonthSchedule

```
public class MonthSchedule extends BasicSchedule {

	@Override
	protected Date getNext(Calendar c1) {
		if (constraint == null)
			return super.getNext(c1);
		else {
			c1.add(Calendar.MONTH, getStep());
			adjustDate(c1);
			return c1.getTime();
		}
	}

	private void adjustDate(Calendar c1) {
		MonthConstraint dc = (MonthConstraint) constraint;
		int inst = dc.getInstance();
		c1.set(Calendar.DAY_OF_MONTH, inst);
		Date now = new Date();
		if(c1.getTime().before(now)) {
			c1.add(Calendar.MONTH, getStep());
			adjustDate(c1);
		}
		this.start = c1.getTime();
	}

	@Override
	public void setConstraint(Constraint constraint) {
		this.constraint = constraint;
		Calendar c = Calendar.getInstance();
		c.setTime(start);
		adjustDate(c);
	}
}
```

MonthConstraint

```
public class MonthConstraint implements Constraint {

	private int instance;

	public MonthConstraint(int instance) {
		this.instance = instance;
	}

	public int getInstance() {
		return instance;
	}

	@Override
	public ScheduleConstraint getType() {
		return Schedule.ScheduleConstraint.MonthConstraint;
	}
}
```

3. Code Smells and System Level Refactoring

By examining our systems' architecture we have found God Class smell. As well as we have found feature Envy, Long Method, Duplicate Code and Collapsible If Statements smells in our project. These are the four smells which we are going to solve in our project so we will explain in detail about this smells in next segment. God Class smell is higher level smell and it requires system level refactoring. Following section will explain about God Class smell and how we can apply different refactoring technic to solve this big smell.

God Class: In object oriented programming language god class is one that knows too much or does too much. In other words it knows about everything. We have found one class by examine our architecture and that class is **FManEntityManager**. It contains too many variables, methods and responsibility which are sign of lazy programming style which results in higher maintenance cost and effort later on.

FManEntityManager contains too many methods and manipulates too much data so it knows about too many object and other object have to depend on this class for the information or interaction. Other object tightly depended on that god class so that other object will not communicate with each other for most of information. Since God class is tightly coupled with other code it is very hard to maintain that class.

It contains responsibilities like adding, deleting and updating of account , It also gives net worth history like report of certain time period, it also add, delete transaction it also update, add account number it can add, delete and update portfolio.

We can refactor this smell by Extracting Class from **FManEntityManger** and we can reduce some responsibilities of that class. After refactoring we will have one another class which will have some of the responsibilities of older class. For solving God Class smell we will use Move field, Move method, extract class refactoring technics.

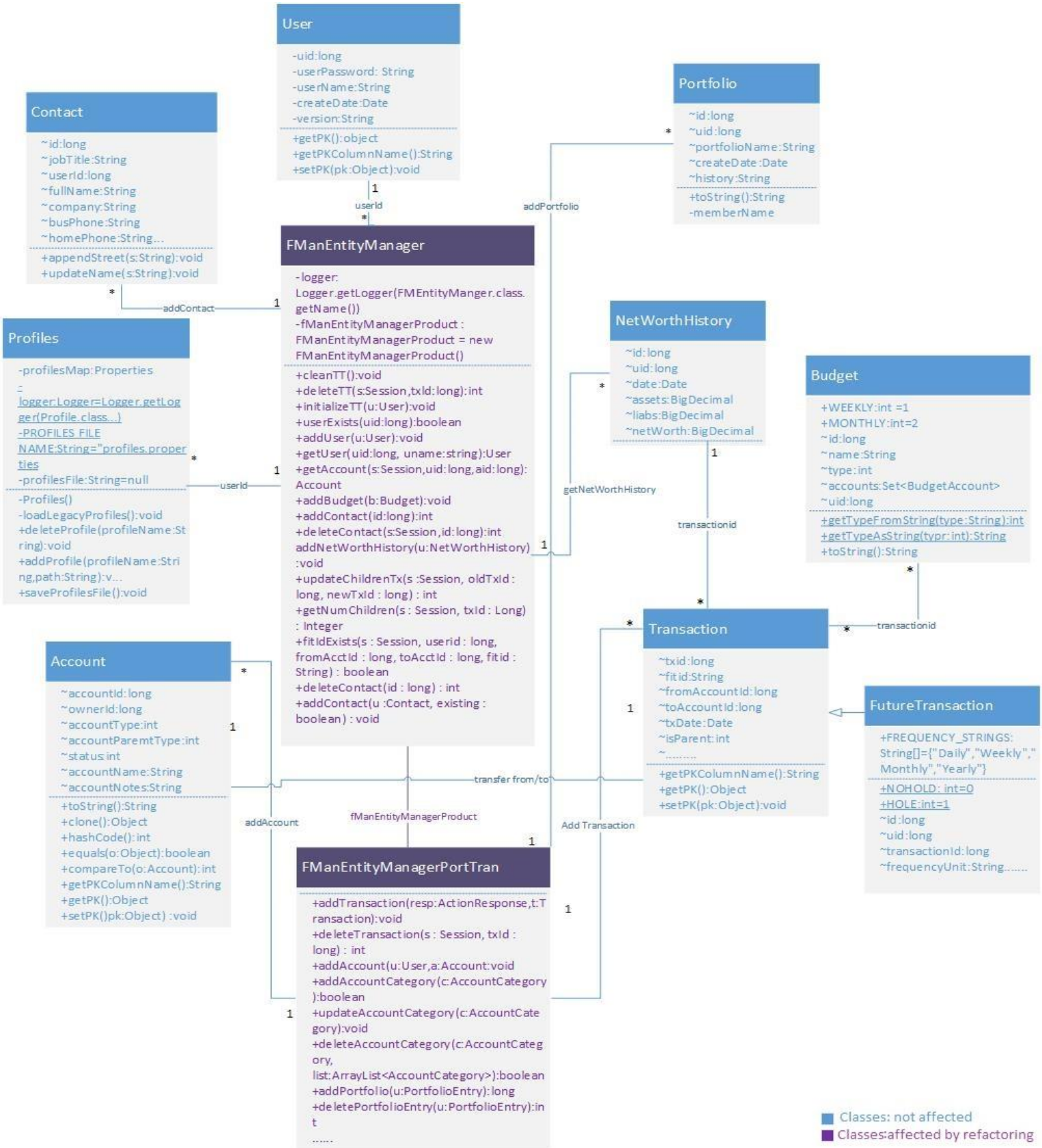
We can do this refactoring by following steps:

- We will decide how to extract some of responsibilities of that class
- We will create new class called **FManEntityManagerTranPort** to transfer some of responsibilities of older class to new class.
- We will change name of the older class if it no longer match its name due to its limited responsibilities.
- We will create field holding reference to **FManEntityManagerTranPort**in older class.
- We will use "Move Field" refactoring technic to move field from older class to new class.
- We will compile and test after every move.
- We will use "Move Method" refactoring technic to move methods like add transaction and delete transaction etc. By this we will increase cohesion and reduce coupling.
- We will compile test after every move.
- We will finally test the whole system to ensure that behavior of system is not changed.

After the class has been extracted, and we've moved the relevant responsibility into the respective areas, we may also need to use the following refactoring technic in order to maintain our design goals: Rename Method, Hide Method, and Replace Parameter with Method. This refactoring will make the architecture more cohesive, and assign responsibility to where it is best suited. So that to solve higher level smells like God class we have to apply series of refactoring technic in order to solve it perfectly.

As we can see in below diagram after refactoring we will have another class called **FManEntityManagerTranPro**. It will have responsibilities to do function about transaction portfolio and account. Some of the method will be moved from **FManEntityManager** to **FManEntityManagerTranPro** and which are **addAccount, deleteAccount, addtransaction, addportfolio** etc. After refactoring **FManEntityManager** has fewer responsibilities then before so now it don't have to worry about transaction, account or portfolio class. Account, transaction and portfolio details will be updated by **FManEntityManagerTranPro** class. We have showed refactored part with purple colour in diagram.

Diagram (After Refactoring):



4. Specific Refactorngs that we will implement in Milestone 4

Code Smells:

1) **Feature Envy**: It is smell about method in one class is more interested in other class than the one it actually is in. We have found this smell in **FManEntityManager** class. It contains method called **addNetWorthHistory** which is actually interested in class called **NetWorthHistory**. So we are going to move that method from **FManEntityManager** to **NetWorthHistory**. We will use move method refactoring technic to solve this smell.

2) **Long Method:** Longer the method is harder to understand it. Long method smell is solved by Extract Method. We have find Long Method smell in **FManEntityManager** class so we will solve it by extracting method. We will find some part of code that will grouped together and make a new method.

3) **Duplicate code/Similar Code:** This smell is about similar code some class or method contains exact same code. We can solve it by pull up method. We have to extract method and use that method in different places by giving reference of that. We found this smell in **AddTransactionAction** and **AddNestedTransactionAction** classes. We can solve it by removing duplicate code from one class and invoking that code by giving references in another class.

4) **Collapsible If Statements:** Sometimes two consecutive 'if' statements can be consolidated by separating their conditions with a Boolean short-circuit operator. We can solve this smell by applying Consolidate Conditional Expression. In our project we have find too many smells like this.

Possible Refactorings:

1) **Move Method:** We will use this refactoring technic to solve feature envy smell which we found in **FManEntityManager** class. We will solve it by following steps:

   - We will find some code that we want to move to another class. Like we will find one method called **addNetWorthHistory.**
   - We will create method in target class called **NetWorthHistory**
   - We will copy code of **addNetWorthHistory** from source class called **FManEntityManager** to target class called **NetWorthHistory.**
   - We will compile target class **NetWorthHistory.**
   - We will leave delegate to **addNetWorthHistory.** We will make **addNetWorthHistory** into delegating method in **FManEntityManager** class.

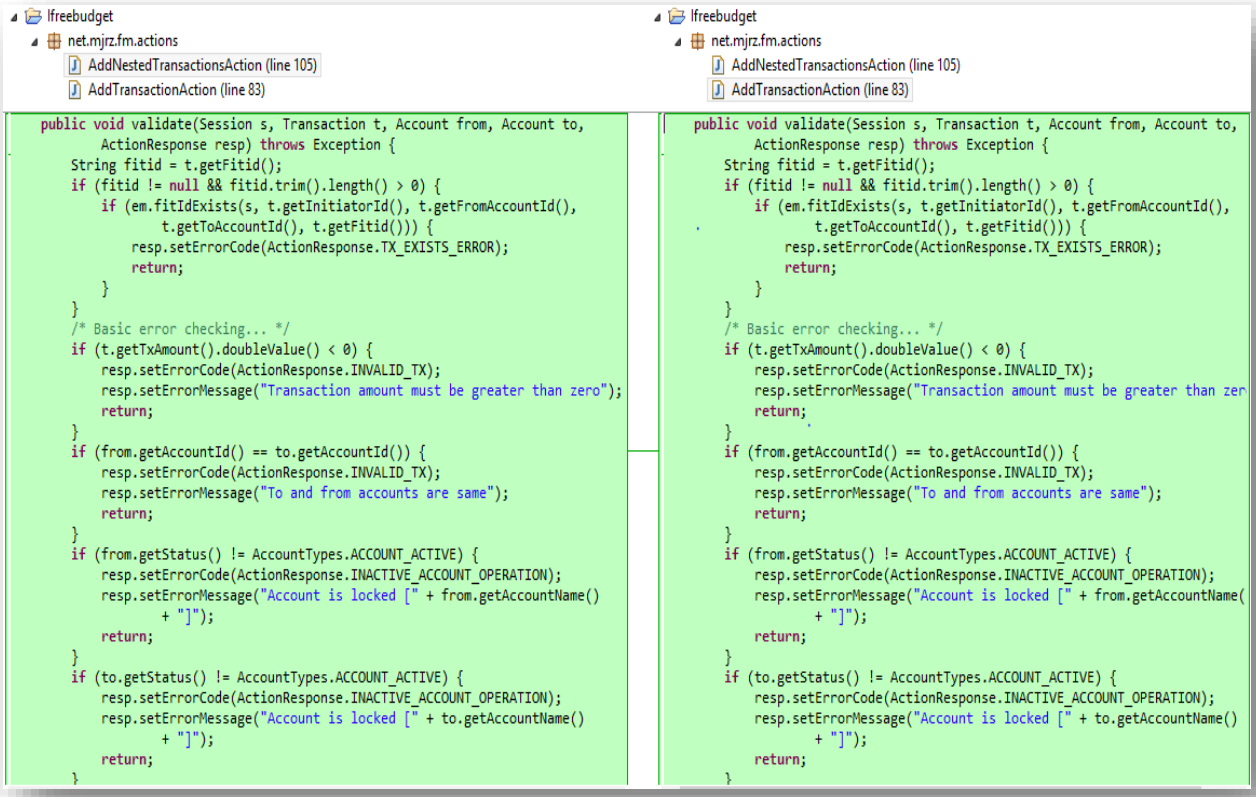
- We will compile and test whole system to ensure behaviour of system is not changed.

2) **Extract method:** This refactoring technic we will use to solve code smell called long method which we found in **FManEntityManager** class**.** We will find some code that we can group together and we will create new method by that code. This refactoring technic is used when we have long method and it is hard to understand we will group some of code that can be fit together and we will give appropriate name to the extracted method according to functionality of that code. We have too many long methods in **FManEntityManager** that can be extracted and we can create a new method from that. We will group some code from that class from particular long method and we will copy it in new method and will give reference to that new method in older method so it can invoke that new method. After completion of that we will compile and test to ensure that behaviour of system is not changed.

3) **Consolidate Conditional Expression:** This refactoring technic is used to solve some complex if statements or when we have sequence of if statement which gave same tests result so we can combine this condition using Boolean operators. Even we can extract that part of code and create new method and invoke that new method in actual test condition. We have found some of this smells in our project and we will solve it by applying Consolidate Conditional Expression refactoring technic.

   Sample class to be refactored:

   This is the snippet of duplicate code of two different classes. One class is AddNestedTransactionAction and other one is AddTransactionAction. Both the classes have method called validate(). We will solve it by removing similar code from any of class and giving reference of another class' method in that class or we will pull up method if they are subclasses of one super class. We have found this smell via Google code pro analytic tool.




```
public void validate(Session s, Transaction t, Account from, Account to,
        ActionResponse resp) throws Exception {
    String fitid = t.getFitid();
    if (fitid != null && fitid.trim().length() > 0) {
        if (em.fitIdExists(s, t.getInitiatorId(), t.getFromAccountId(),
                t.getToAccountId(), t.getFitid())) {
            resp.setErrorCode(ActionResponse.TX_EXISTS_ERROR);
            return;
        }
    }
    /* Basic error checking... */
    if (t.getTxAmount().doubleValue() < 0) {
        resp.setErrorCode(ActionResponse.INVALID_TX);
        resp.setErrorMessage("Transaction amount must be greater than zero");
        return;
    }
    if (from.getAccountId() == to.getAccountId()) {
        resp.setErrorCode(ActionResponse.INVALID_TX);
        resp.setErrorMessage("To and from accounts are same");
        return;
    }
    if (from.getStatus() != AccountTypes.ACCOUNT_ACTIVE) {
        resp.setErrorCode(ActionResponse.INACTIVE_ACCOUNT_OPERATION);
        resp.setErrorMessage("Account is locked [" + from.getAccountName()
                + "]");
        return;
    }
    if (to.getStatus() != AccountTypes.ACCOUNT_ACTIVE) {
        resp.setErrorCode(ActionResponse.INACTIVE_ACCOUNT_OPERATION);
        resp.setErrorMessage("Account is locked [" + to.getAccountName()
                + "]");
        return;
    }
```

```
public void validate(Session s, Transaction t, Account from, Account to,
        ActionResponse resp) throws Exception {
    String fitid = t.getFitid();
    if (fitid != null && fitid.trim().length() > 0) {
        if (em.fitIdExists(s, t.getInitiatorId(), t.getFromAccountId(),
                t.getToAccountId(), t.getFitid())) {
            resp.setErrorCode(ActionResponse.TX_EXISTS_ERROR);
            return;
        }
    }
    /* Basic error checking... */
    if (t.getTxAmount().doubleValue() < 0) {
        resp.setErrorCode(ActionResponse.INVALID_TX);
        resp.setErrorMessage("Transaction amount must be greater than zer
        return;
    }
    if (from.getAccountId() == to.getAccountId()) {
        resp.setErrorCode(ActionResponse.INVALID_TX);
        resp.setErrorMessage("To and from accounts are same");
        return;
    }
    if (from.getStatus() != AccountTypes.ACCOUNT_ACTIVE) {
        resp.setErrorCode(ActionResponse.INACTIVE_ACCOUNT_OPERATION);
        resp.setErrorMessage("Account is locked [" + from.getAccountName(
                + "]");
        return;
    }
    if (to.getStatus() != AccountTypes.ACCOUNT_ACTIVE) {
        resp.setErrorCode(ActionResponse.INACTIVE_ACCOUNT_OPERATION);
        resp.setErrorMessage("Account is locked [" + to.getAccountName()
                + "]");
        return;
    }
```