Functional Requirements to Software System using Object Oriented Programming

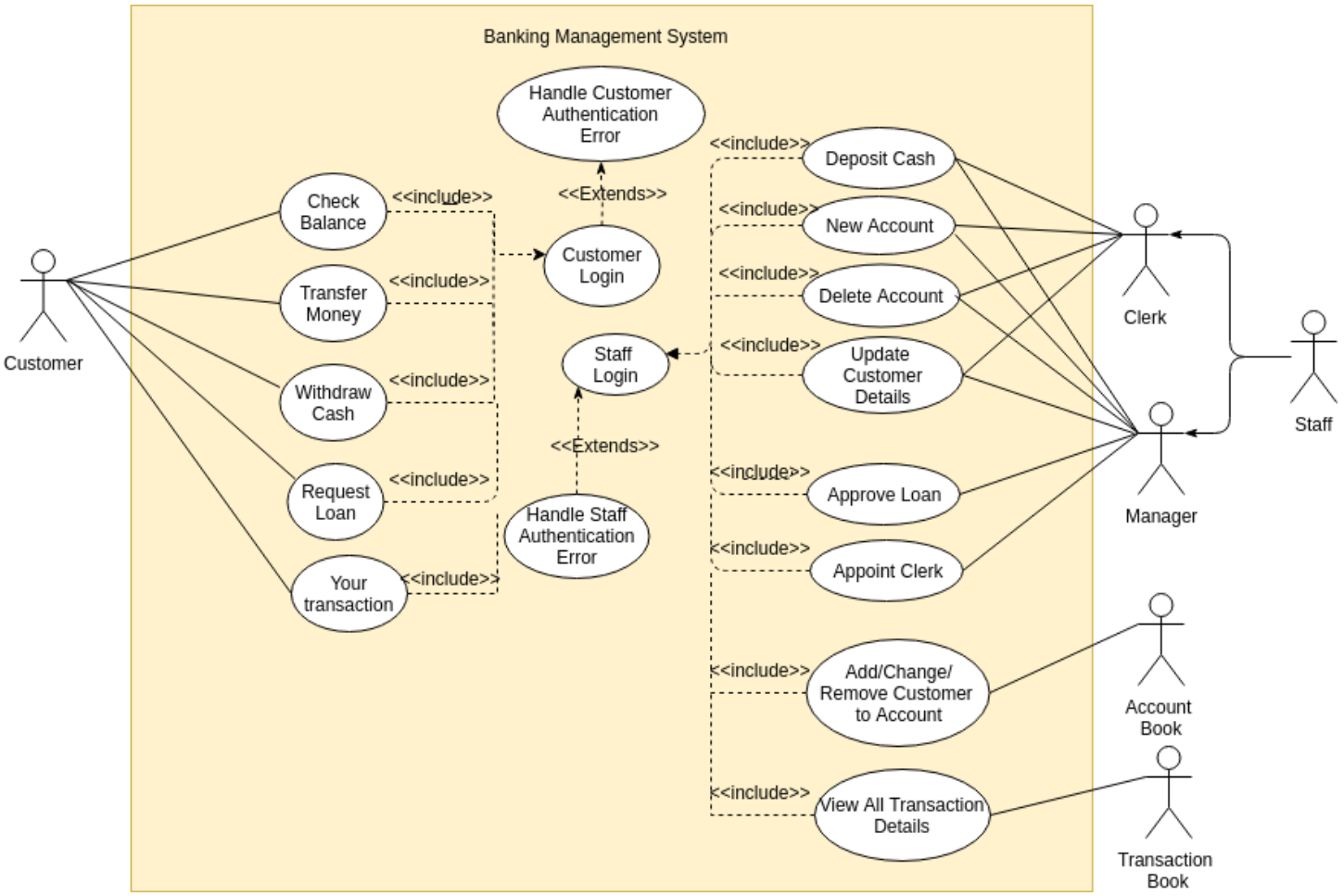
Published on November 15, 2019 Edit article | View stats

In object oriented programming approach we are trying to simulate or operate the real world entities using a computer program. It is mainly used for systems where end-user of the product interact with the software which means all mobile apps, websites and enterprise software consider object oriented programming approach to build the main software system. These kind of systems need a software which is portable, secure and easy to distribute so that they don't need technical support to install and maintain it on different machines. To build such software systems the first approach in implementation of programming language called ***C with objects*** later known as ***C++*** was made by a famous Danish scientist Bjarne Straustrup in 1979 during his PhD thesis. The problem with C++ was it's incapability to operate properly on different machines which means it run properly on some systems, produce bug on some system and sometimes it crash as well. So we need to consider different code for different machines. To overcome this flaw, James Gosling along with his two friends Mike and Patrick initiated the ***Java*** language project at Sun Micro-systems in 1991. The team expanded and develop the concept of a virtual machine and built ***Java Virtual Machine***, a software which behaves like a machine also known as virtual machine. The team built different JVM software for different machines so that programmers don't have to worry about anything regarding machine configurations or which machine will execute my Java code. If machine has JVM installed then we don't have to worry about our code to optimise it properly for different machine which make Java ***platform independent or architecture*** . This concept revolutionaries the whole industry because most of the

companies wants to automate their real world process using computer software in order to boost their productivity using computer's efficiency.

Let me take an example of ***Banking management system*** for the demonstration of implementation of real world system in the computer using object oriented programming technique. Earlier, banks use registers, files and record books to perform banking operations where we need to rely on ***clerks*** for everything because we don't have a secure and efficient way to provide access of records to the customers. *Basically, a bank receive money from some customers and provide to other customers who need it as loan with some interest in order to benefit all stake holders who invest their money in the bank from customers, employees to share holders*. In a bank we have ***customer***, ***banking staff*** and ***record books*** as the main components. To build a bank we need to understand the functional requirement of the bank or what are the functionality of each user or entity of the bank in the banking system.

Customer of the bank can *check his account balance, withdraw cash, transfer money, request loan* and *check his/her transaction details*. Staff member of the bank can *deposit cash, create and delete account, change account ownership, approve loan and view all transaction details.* Transaction book keep the *records of all the transactions* happened in the bank. We can use UML (Unified Modelling Language) document like *use case diagram* to specify our requirements to build the desired software.



To represent a customer in the software system we need to first define the properties and behaviour of the customer in the context of banking system. ***Our customer can eat, read, sleep, play and eat but these details are irrelevant for banking operations so we need to define our blueprint of customer strictly in the context of banking system***. ***Context is the most important thing while creating any system software*** because creating a highly detailed document about the personal preferences of customers to perform some banking operations isn't really valuable for the banking system. Security guard of your college isn't interested to ask you what you ate in the breakfast, he just want your college ID card. So, we have no interest to provide full detail about the customer in the software as we are only dealing with a few aspects or context of the customer required by the system. ***Whenever we change the context or perspective of the customer we need to change or modify the blueprint of the***

customer in the software which means ***it is impossible to create a complete software so* our *target should be to create a working software. Our whole purpose to build system software is to improve the business operations.***

To create the blueprint of the customer in the program we will use **class** in Java.

```
// Way to define a blueprint using Java class
// In our case we are defining the customer


class Customer{


    int Customer_ID,Account_Number;
    String name, address;

    // this method is known as contructor as it invoke when object is
    // constructed

    public Customer(int c_id, String name, String address, int acc_no){

        this.Customer_ID = c_id;
        this.name = name;
        this.address = address;
        this.Account_Number = acc_no;

    }


}

public class Main{

        public static void main(String[] args) {


        // Way to create the customer as Java object ..

        Customer c = new Customer(1, "Yash", "Ghaziabad", 1122);

                System.out.println("\nCustomer Details\n");

                System.out.println(
                    " Customer_ID : "+ c.Customer_ID +
                    " Name : "+ c.name +
                    " Address :"+ c.address +
                    " Account_Number : "+ c.Account_Number +"\n"
                );


        }
}
```

```

```

Output

```
Customer Details
 Customer_ID : 1 Name : Yash Address :Ghaziabad Account_Number : 1122
```

In Java, members of class are accessible from the outside by default which is similar to ***"clerk give register/file/record book to customer and tell them to fill the entries manually"***. It is not only a bad way to serve customer but also insecure way to perform banking operation, people can change ownership, add any number of zeros in their account balance which makes the whole system corrupt.

To overcome this problem in software we introduced the concept of access modifiers where we can control the access of class members. So we will create our class in such a way that data members can be only accessed by the member function of the class, not anywhere from the outside. Wrapping up data and function in the single unit using class with defined way for access it is also known as ***encapsulation***. We can only turn or change the direction of car using steering provided by the car manufacturer not by any other way. This way to use the object property with well defined methods is known as encapsulation. Other way to think about encapsulation is, ***it is a protective shield that prevents the data from being accessed by the code outside this shield.***

Lets refactor the above code and make it encapsulated by using access modifiers.

```
class Customer{
    // using private keyword we protect the data

    private int Customer_ID,Account_Number;
    private String name, address;

    // We created a 4 digit PIN or password to access and modify details
    private int PIN;

    // set it true if user enter the correct pin
    private boolean allow_access = false;

    public Customer(int c_id, String name, String address, int acc_no){

        this.Customer_ID = c_id;
        this.name = name;
        this.address = address;
        this.Account_Number = acc_no;

    }

   // we don't want to reveal information without correct password

   public void enterPIN(String user_pin){
       this.allow_access = (this.PIN == user_pin);
   }


   // these kind of methods used just to get the values
   // are known as getters

   public int getCustomerID(){
       return this.Customer_ID;
   }


   private String getWrongPINError(){
      return "Please enter correct PIN .. ";
   }


   // show details only if correct pin entered

   public String getDetails(){

     if(this.allow_access == false)
        return getWrongPINError();

     return " Customer ID : " + this.Customer_ID +
            " Name : " + this.name +
            " Address : " + this.address +
            " Account_Number : " + this.Account_Number;
   }


    // allow to set Details only if correct pin entered
    // these kind of methods just used to set the values
    // are known as setters
```

```
    public void setName(String name){
       if(this.allow_access == false)
         System.out.println(getWrongPINError());

       this.name = name;
    }


    public void setAddress(String address){

       if(this.allow_access == false)
         System.out.println(getWrongPINError());

       this.address = address;
    }


    public void setAccountNo(int AccountNo){

       if(this.allow_access == false)
         System.out.println(getWrongPINError());

       this.Account_Number = AccountNo;
    }

   // resetPIN

    public void resetPIN(int PIN){

       if(this.allow_access == false)
         System.out.println(getWrongPINError());

       this.PIN = PIN;
    }

    // Log Out

    public void logOut(){
       this.allow_access = false;
    }

 }
```

Now, we have created our encapsulated class for customer by making all the member private and allow to change it via member functions only where you need to enter correct PIN before making any changes in name, address, account number etc. This is not possible in our older physical register/file/record book based system to provide access to customer directly and securely without involvement of any clerk. ***This is the real power of computer software which boost the productivity and save the money spent on banking operations.***

A Manager can do whatever a clerk can or we can say a manager inherit all functionality of the clerk in the system. This type of relationship is implemented by keyword extends and the process is known as ***inheritance***. The inheritance is also known as "is-a" relationship.

```
class Clerk{

   // We use protected keyword we want access Staff_ID in
   // the derived class called manager
   // but we don't want anyone to access it from the outside

   protected int Staff_ID;
   String name, address;

   ..


   public void deposit_cash( int amount, Account account ){

        account.addMoney(amount);

   }

   public void create_new_account( .. ){
     ..
   }


   public void close_account( .. ){
     ..
   }

   public void update_customer_details ( .. ) {
     ..
   }



}

// Manager inherit all features of Clerk or a manager can do
// everything a clerk can do

// final is used to define we can't inherit Manager class

final class Manager extends Clerk{

   ..

   public void approve_loan( .. ){
     ..
   }


   public void appoint_clerk( .. ){
```

```
        ..
    }



}
```

Earlier, In customer class we use account_number to access account object but we can also define account inside the customer class as a member variable. This type of relationship is also known as ***aggregation.*** The aggregation is also known as "has-a" relationship.

```
class Customer{

 int Customer_ID;
 String name, address;

 Account account;  // aggregation to create better blueprint

 ..

}
```

When we directly create the class using the ***use-case diagram*** we need to consider all the function and it's definition while defining the class but in java we can only define the abstract details of class using the concept of ***abstraction***. Abstraction is the process of hiding the implementation details and it can be achieved by ***abstract class*** and ***interface.***

Manager and clerk both are staff member but there is no staff member exist in the banking system, it is just a representation or an abstract concept which can be implemented by using abstract keyword. ***Fruit is an abstract where apple is the real existing entity so we will represent fruit in the system using abstract class***. If you look at the bucket of vegetables you purchase from vegetable stall or in context of vegetable, there is no such thing called vegetable, we will only find something like tomato, potato etc. We will use the extends

keyword to inherit all the feature of fruit when we try to create the class of mango or apple as mango contains all the features of fruits.

```
// we make staff as abstract using "abstract" keyword
abstract class Staff{
   int Staff_ID;
   String name, address;
   ..
}

// Clerk "is-a" staff member

class Clerk extends Staff{
  ..
}


// Manager can behave like a clerk but vice versa isn't possible
// or a manager can do whatever a clerk can do ..

class Manager extends Clerk{
  ..
}

class Main{

  public static void main(){

      Staff s = new Staff(); // it will produce error .. because we can't
                             // create the object of an abstract class
                             // because it doesn't exist in the real system

  }


}
```

We don't know how our customer check balance, transfer money, withdraw cash and view transaction detail from the system but ***we want our customer must have these functionalities if he/she wants to interact or interface with our system properly***. If we miss any detail in our system we will not be able to create the object of class. This concept to define abstract details about the customer is done by using ***interface*** in java. The advantage of such techniques is we can define the interface or the way of interaction using ***use cases*** very conveniently as functions needed to declare inside the interface are already present in the

use case diagram. We don't have to brainstorm anymore about which functionalities I need once we get the use case diagram properly. It boost the productivity of the team working on the same project by removing the communication problem between technical and non technical team members responsible for the product development.

```
// function declaration only .. inside interface .. no definition

interface CustomerFunctionalities{

   void check_balance();
   void transfer_money(int amount, Account account);
   void withdraw_cash(int amount, Account account);
   void view_transaction_details();

}


// here we will define our functions we declared inside interface

class Customer implements CustomerFunctionalities{


   void check_balance(){
     ..
   }

   void transfer_money(int amount, Account account){
     ..
   }

   void withdraw_cash(int amount, Account account){
     ..
   }

   void view_transaction_details(){
     ..

   }


}
```

In our bank there is no such thing called account but we have business account, current account and saving account and each had different loan interest rate so we will **override** these functions in the derived class so that we don't have to use different function names to

ask the system interest rate for different types of account. This concept is known as ***polymorphism*** where one name is used to perform different functionality. We can also understand it like dear, lion and dog are three different animal (animal is abstract where dear, lion and dog are real entities) but ***they perform the same function but differently*** say eat then each one of them eat food but they eat differently lion eat dear, dear eats plants and dog eats bread. Using polymorphism we just reduce the complexity of code otherwise we need to define it like dear_eat(), lion_eat() and dog_eat() but now using polymorphism we can simply use eat() it will refer the function according to the object of animal. If we know the animal is lion, compiler will know he eats deer as food. This is also known as run-time or dynamic polymorphism.

```
// In banking system we don't have anything called Account
// it is abstract ..

abstract class Account{

   int getLoanInterestRate(){ return 15; }

}

class BusinessAccount extends Account{

   int getLoanInterestRate(){ return 20; }

}

class CurrentAccount extends Account{

  int getLoanInterestRate(){ return 18; }

}

class SavingAccount extends Account{

  int getLoanInterestRate(){ return 16; }

}

class Main{

 public static void main(){

    Account a1 = new BusinessAccount();
    System.out.println("Loan Rate : "+a1.getLoanInterestRate());

    Account a2 = new SavingAccount();
    System.out.println("Loan Rate : "+a2.getLoanInterestRate());
```

```
     Account a3 = new CurrentAccount();
     System.out.println("Loan Rate : "+a3.getLoanInterestRate());


 }

}
```

Output

```
 Loan Rate : 20
 Loan Rate : 16
 Loan Rate : 18
```

Bank will charge Rs. 50 on every transaction made with business account, Rs. 30 with saving account and Rs. 40 with current account. We can do it with the help of method ***overloading*** in the computer program. This is also known as compile time polymorphism

```
class Customer implements CustomerFunctionalities {

   ...


   // same name different functionality but using different
   // function signature or using different function definition
   //

   int getServiceCharge(BusinessAccount){
       return  50;
   }

   int getServiceCharge(CurrentAccount){
       return  40;
   }


   int getServiceCharge(SavingAccount){
       return  30;
   }


}
```

So we finally learnt about the main features like encapsulation,inheritance abstraction and polymorphism of object oriented programming. Now using all the above knowledge we need to draw the class diagram for the given use case diagram to create the software. ***There is no right or wrong class diagram or software, it is just about more or less efficient implementation which determine the productivity and easy to maintenance of the given system.***