Software Design / Programming

An Implementation Of

The Proxy Pattern



Student Name : David Martin

Student Number : A00128473

MSc. Software Engineering

Table of Contents

[Project Proposal 3](#_Toc428124617)

[Introduction 4](#_Toc428124618)

[A definition of the word “Proxy” 4](#_Toc428124619)

[What is the Proxy Pattern? 4](#_Toc428124620)

[Structure of the Proxy Pattern 5](#_Toc428124621)

[Proxy Pattern Example 6](#_Toc428124622)

[Class Diagram of the ATM Proxy Pattern Example 6](#_Toc428124623)

[Code Explanation of the ATM Proxy Pattern Example 7](#_Toc428124624)

[Bank Class (Real-Subject) 7](#_Toc428124625)

[Proxy Class (Proxy) 8](#_Toc428124626)

[ATM Interface (Subject) 9](#_Toc428124627)

[Client 9](#_Toc428124628)

[Conclusion 10](#_Toc428124629)

[References 10](#_Toc428124630)

[Source Code 11](#_Toc428124631)

[MainApp.cs 11](#_Toc428124632)

[Account.cs 13](#_Toc428124633)

# Project Proposal

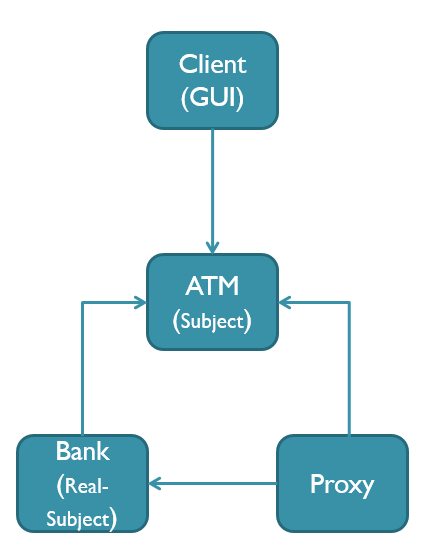
I intend to design a Proxy Pattern that will allow a Client, by the use of the ATM Interface, to fetch the user’s bank account details from the Bank Class & perform basic operations on their account upon authentication via the use of a protection proxy.

The proxy will be implemented to allow the Client to request only certain operations from the bank class, via the ATM Interface, upon the User’s login credentials being authenticated by the Proxy. Hence demonstrating the Proxy Patterns ability to provide an ***additional security layer***, ***restricted access*** to the functionality of the Bank Object via the use of the Interface (the Subject) and the **“*lazy instantiation*”** of the Bank Object (the Real Object) via the Proxy.

My goal is to add the protection proxy object as a layer between the ATM Interface code and the Bank Class logic.

The Bank Object will contain an Array of Bank Account Details and some simple operation (getter and setter) methods. Limited access to these operations will be restricted by the ATM Interface to allow the User to withdraw/desposit funds from his/her bank account & check current balance of the account via the Proxy.

Ultimately, The Proxy layer implemented in this program will provide a surrogate or placeholder for the Bank object, also known as the Real Subject, which in turn will provide the ability to control access to the Bank object.



*ATM Class Structure of Proxy Pattern Implementation*

A working demonstration of this implementation will be shown via the use of a Console application.

# Introduction

## A definition of the word “Proxy”

The word “Proxy” originates from the Medieval Latin term “procuratia” which simply means “administration”. It can also be associated with the early 15th century terms “proccy” or “prokecye” which means "agency of one who acts instead of another; letter of power of attorney".

## What is the Proxy Pattern?

According to the Gang of Four's, Design Patterns “Elements of Re-useable Object Orientated Software” book, the Proxy Pattern is defined as follows: "The Proxy Pattern provides a surrogate or placeholder for another object to control access to it."

This design pattern can be used for various reasons as follows:

* implementation of Security,
* because an Object could be resource costly/intensive to create
* accessing objects from a remote location

You can think of it as a gate keeper that blocks/controls access to another Object.

Proxy pattern comes into play when we have resource expensive objects and

* We want to implement a simpler version of a resource costly object.
* We don’t need the whole functionality of the resource expensive object.
* We want to limit the access to the resource expensive object.
* Because there may be a time-delay or complex mechanism in creating instances of HeavyWeight objects.

Proxy pattern is also known as surrogate. This means to put something into place of another.

To help explain the purpose of the Proxy Pattern, the following is a real-world example.

In this hypothetical situation, let’s say that there is a third party program whose purpose is to get the current trading price of Oil from the NASDAQ Index on the US Stock Exchange. These prices will not be made available for the third party program directly. As such, the application must make use of a proxy class to get the actual price quotes, as the main system used by the NASDAQ, responsible for providing this information, will not allow any third-party to access its database directly.

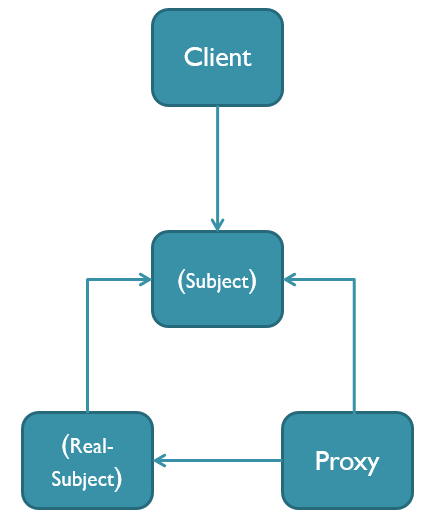
In this case the proxy will fetch the current prices from the actual source, only when requested by the third party program. So the program code will be calling the proxy class and the proxy class will get the price from the actual source and send the current price, back to the third party program in question.

From this example, we can see that the Proxy Pattern allows not only to control creation/access to the resources only when required, but also provides security as the client code cannot directly access the resources and must pass through the proxy.

Proxies can provide additional functionality depending on the situation, for example:

* **remote proxies** are responsible for encoding a request and its arguments and for sending the encoded request to the real subject in a different address space.
* **virtual proxies** may cache additional information about the real subject so that they can postpone accessing it.
* **protection proxies** check that the caller has the access permissions required to perform a request.

## Structure of the Proxy Pattern

****

*UML Diagram of the Proxy Pattern Structure*

The classes and objects that create the structure of this pattern are as follows:

**Proxy**

* + maintains a reference that lets the proxy access the real subject.
  + provides an interface identical to Subject's with extra functionality if required, eg: security features, so that a proxy object can be substituted for the real subject.
  + controls access to the real subject and may be responsible for creating and deleting instances of it.

**Subject**

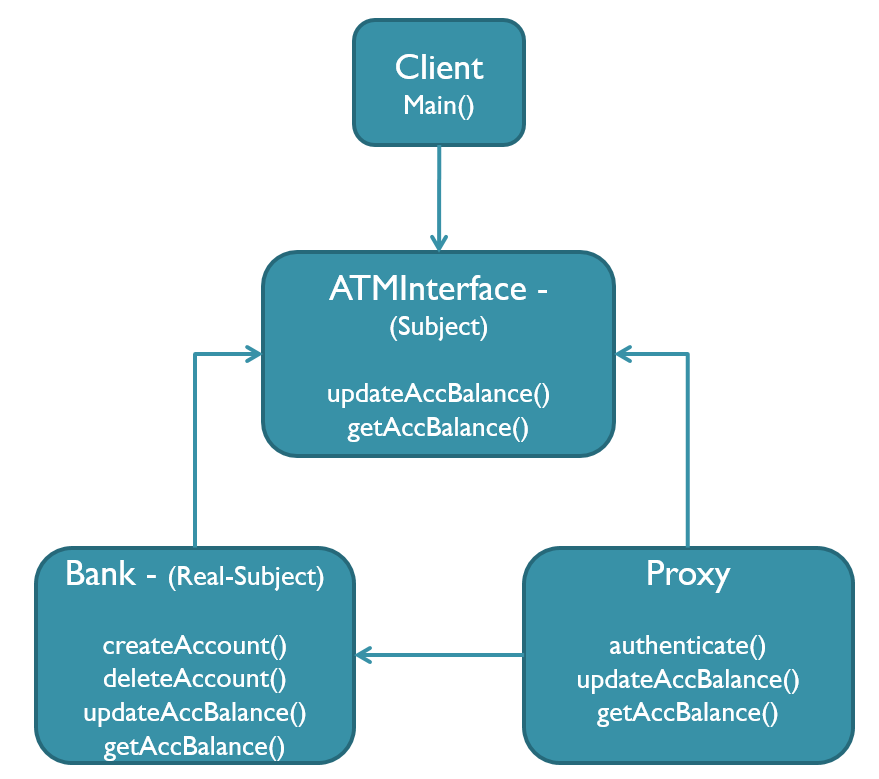
* + defines the common interface for RealSubject and Proxy so that a Proxy can be used anywhere a RealSubject is expected.

**RealSubject**

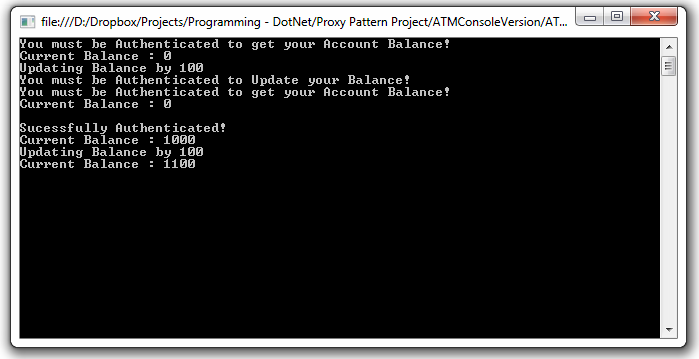
* + defines the real object that the proxy represents.

# Proxy Pattern Example

## Class Diagram of the ATM Proxy Pattern Example

**

*Methods of ATM Proxy Pattern implementation*

****

*Execution of ATM Proxy Pattern Example*

## Code Explanation of the ATM Proxy Pattern Example

### Bank Class (Real-Subject)

The Bank class acts as the Real Subject in this Proxy Example. This class provides various methods with allow for the creation and deletion of bank accounts, the logic to update a bank account balance and also return the current balance of a bank account. For the sake of demonstration we will assume that the creation of a Bank object is costly in terms of resources.

The constructor creates numerous Account objects and adds them to a List of Account objects called accountArray. For the createAccount(), deleteAccount(), UpdateAccBalance() & getAccBalance, the use of a LINQ Lambda expression is used to search the List for the account specified for which the action in question will be implemented. The Bank Class extends the ATMInterface class.

// The 'RealSubject' class

class Bank : ATMInterface

{

private List<Account> accountArray = new List<Account>();

private int NumOfAccounts = 0;

int accountIndex;

public Bank()

{

accountArray.Add(new Account(100, 1000.00, "Dave Martin", "Athlone",

"Personal", "Good"));

accountArray.Add(new Account(101, 1000.00, "Joe Bloggs", "Galway",

"Corporate", "Good"));

accountArray.Add(new Account(102, 1000.00, "Jane Bloggs", "Dublin",

"Personal", "Good"));

accountArray.Add(new Account(103, 1000.00, "John Doe", "Waterford",

"Corporate", "Good"));

NumOfAccounts = 4;

}

public void createAccount(int a, double b, string c, string d, string e,

string f)

{

accountArray[NumOfAccounts] = new Account(a, b, c, d, e, f);

NumOfAccounts++;

}

public void deleteAccount(int a)

{

accountIndex = accountArray.IndexOf(accountArray.

Where(p => p.readAccNo() == a).FirstOrDefault());

accountArray.RemoveAt(accountIndex);

NumOfAccounts--;

}

public void updateAccBalance(int a, double b)

{

accountIndex = accountArray.IndexOf(accountArray.

Where(p => p.readAccNo() == a).FirstOrDefault());

double newBalance = accountArray[accountIndex].readBal() + b;

accountArray[accountIndex].setBal(newBalance);

}

public double getAccBalance(int a)

{

accountIndex = accountArray.IndexOf(accountArray.

Where(p => p.readAccNo() == a).FirstOrDefault());

return accountArray[accountIndex].readBal();

}

}

### Proxy Class (Proxy)

The Proxy class provides an “***additional security layer***” via the implementation of the authenticate() method, whose purpose is to authenticate the customers login credentials before any interaction can be performed with the real-subject. The use of “***lazy instantiation***” of the Bank Object (the Real Subject) is also implemented in this Proxy pending a valid authentication form the Client.

// The 'Proxy' class

class Proxy : ATMInterface

{

Bank myBank;

public void authenticate(string a, string b)

{

if (a == "myUsername" && b == "myPassword")

{

Console.WriteLine("Sucessfully Authenticated!");

myBank = new Bank();

}

}

public void updateAccBalance(int a, double b)

{

if (myBank == null)

{

Console.WriteLine("You must be Authenticated to Update your

Balance!");

}

else

{

myBank.updateAccBalance(a, b);

}

}

public double getAccBalance(int a)

{

if (myBank == null)

{

Console.WriteLine("You must be Authenticated to get your Account Balance!");

return 0.0;

}

return myBank.getAccBalance(a);

}

}

The authenticate() method is used to validate the login credentials of the bank customer by use of an If statement. Only when login credentials, supplied by the User, which meet the conditions of this If statement shall a Bank Object be created. All other methods within this class will check to ensure that the Bank Object has being instantiated and (eg: the myBank reference to the Bank Object is not null), which is ultimately determined by a successful login, before their logic will be executed.

### ATM Interface (Subject)

The ATM interface will act as the Subject in this example, providing an interface for the client to the real subject. The interface will only allow “***restricted access”*** to the functionality of the Bank Class. As depicted in the code example below, this interface will only allow access to two of the methods available in the Bank (Real-Subject) class. Namely, the updateAccBalance() method and the getAccBalance() method.

// The 'Subject interface

public interface ATMInterface

{

void updateAccBalance(int a, double b);

double getAccBalance(int a);

}

### Client

For demonstration purposes, the client in this case will simply consist of a Main() method which will firstly attempt to execute various actions on a bank account after providing invalid login credentials to the authenticate() method. As expected, no communication with the Real Subject will take place as determined by the Proxy class.

On the second attempt, the client will call the authenticate() method with valid login credentials and attempt to execute the same actions as before on a bank account. In this case, the Proxy allows for the creation of a Bank Object and the logic for each subsequent method call thereafter will be acted upon.

Hence, demonstrating a simple implementation of the Proxy Pattern.

// The 'Client' class

class Client

{

static void Main()

{

Proxy myProxy = new Proxy();

// Invalid Login Credentials

myProxy.authenticate("Invalid", "Credentials");

Console.WriteLine("Current Balance : " + myProxy.getAccBalance(100));

Console.WriteLine("Updating Balance by 100 ");

myProxy.updateAccBalance(100, 100);

Console.WriteLine("Current Balance : " + myProxy.getAccBalance(100));

Console.WriteLine();

// Valid Login Credentials

myProxy.authenticate("myUsername", "myPassword");

Console.WriteLine("Current Balance : " + myProxy.getAccBalance(100));

Console.WriteLine("Updating Balance by 100 ");

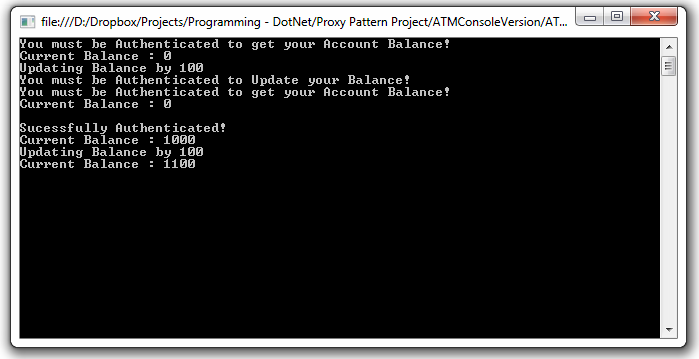
myProxy.updateAccBalance(100, 100);

Console.WriteLine("Current Balance : " + myProxy.getAccBalance(100));

Console.ReadLine();

}

}

****

*Execution of ATM Proxy Pattern Example*

# Conclusion

In this Assignment, I first set out to explain exactly what the Proxy Pattern is, its advantages and the different purposes and features it can provide on implementation. I then created my own hypothetical situation in the form of an ATM Banking System to demonstrate a simple implementation of the Proxy Pattern in action.

Notable features that are demonstrated in my implementation are the Proxy Patterns ability to provide an additional security layer, restricted access to the functionality Real Object) via the Proxy and the of “lazy instantiation” of the Bank Object (the Real Subject).

# References

For research of this paper I studied the following books:

**Design Patterns: Elements of Reusable Object-Oriented Software** written by the Gang of Four, namely, Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides.

**Head First Design Patterns** written by Eric Freeman, Elisabeth Robson, Bert Bates, Kathy Sierra.

For research in the origins and meaning of the word “Proxy” I used the online Etymological Dictionary of Modern English found at the following url : [http://www.etymonline.com/](http://www.etymonline.com/index.php)

# Source Code

## MainApp.cs

**using** System**;**

**using** System**.**Collections**.**Generic**;**

**using** System**.**Linq**;**

**using** System**.**Text**;**

**using** System**.**Threading**.**Tasks**;**

class MainApp

**{**

// The 'Subject interface

**public** **interface** ATMInterface

**{**

void updateAccBalance**(**int a**,** double b**);**

double getAccBalance**(**int a**);**

**}**

// The 'RealSubject' class

class Bank **:** ATMInterface

**{**

**private** List**<**Account**>** accountArray **=** **new** List**<**Account**>();**

**private** int NumOfAccounts **=** 0**;**

int accountIndex**;**

**public** Bank**()**

**{**

accountArray**.**Add**(new** Account**(**100**,** 1000.00**,** "Dave Martin"**,** "Athlone"**,** "Personal"**,**

"Good"**));**

accountArray**.**Add**(new** Account**(**101**,** 1000.00**,** "Joe Bloggs"**,** "Galway"**,** "Corporate"**,**

"Good"**));**

accountArray**.**Add**(new** Account**(**102**,** 1000.00**,** "Jane Bloggs"**,** "Dublin"**,** "Personal"**,**

"Good"**));**

accountArray**.**Add**(new** Account**(**103**,** 1000.00**,** "John Doe"**,** "Waterford"**,** "Corporate"**,**

"Good"**));**

NumOfAccounts **=** 4**;**

**}**

**public** void createAccount**(**int a**,** double b**,** string c**,** string d**,** string e**,** string f**)**

**{**

accountArray**[**NumOfAccounts**]** **=** **new** Account**(**a**,** b**,** c**,** d**,** e**,** f**);**

NumOfAccounts**++;**

**}**

**public** void deleteAccount**(**int a**)**

**{**

accountIndex **=** accountArray**.**IndexOf**(**accountArray**.**Where**(**p **=>** p**.**readAccNo**()** **==**

a**).**FirstOrDefault**());**

accountArray**.**RemoveAt**(**accountIndex**);**

NumOfAccounts**--;**

**}**

**public** void updateAccBalance**(**int a**,** double b**)**

**{**

accountIndex **=** accountArray**.**IndexOf**(**accountArray**.**Where**(**p **=>** p**.**readAccNo**()** **==**

a**).**FirstOrDefault**());**

double newBalance **=** accountArray**[**accountIndex**].**readBal**()** **+** b**;**

accountArray**[**accountIndex**].**setBal**(**newBalance**);**

**}**

**public** double getAccBalance**(**int a**)**

**{**

accountIndex **=** accountArray**.**IndexOf**(**accountArray**.**Where**(**p **=>** p**.**readAccNo**()** **==**

a**).**FirstOrDefault**());**

**return** accountArray**[**accountIndex**].**readBal**();**

**}**

**}**

// The 'Proxy' class

class Proxy **:** ATMInterface

**{**

Bank myBank**;**

**public** void authenticate**(**string a**,** string b**)**

**{**

**if** **(**a **==** "myUsername" **&&** b **==** "myPassword"**)**

**{**

Console**.**WriteLine**(**"Sucessfully Authenticated!"**);**

myBank **=** **new** Bank**();**

**}**

**}**

**public** void updateAccBalance**(**int a**,** double b**)**

**{**

**if** **(**myBank **==** **null)**

**{**

Console**.**WriteLine**(**"You must be Authenticated to Update your Balance!"**);**

**}**

**else**

**{**

myBank**.**updateAccBalance**(**a**,** b**);**

**}**

**}**

**public** double getAccBalance**(**int a**)**

**{**

**if** **(**myBank **==** **null)**

**{**

Console**.**WriteLine**(**"You must be Authenticated to get your Account Balance!"**);**

**return** 0.0**;**

**}**

**return** myBank**.**getAccBalance**(**a**);**

**}**

**}**

// The 'Client' class

class Client

**{**

static void Main**()**

**{**

Proxy myProxy **=** **new** Proxy**();**

myProxy**.**authenticate**(**"Invalid"**,** "Credentials"**);**

Console**.**WriteLine**(**"Current Balance : " **+** myProxy**.**getAccBalance**(**100**));**

Console**.**WriteLine**(**"Updating Balance by 100 "**);**

myProxy**.**updateAccBalance**(**100**,** 100**);**

Console**.**WriteLine**(**"Current Balance : " **+** myProxy**.**getAccBalance**(**100**));**

Console**.**WriteLine**();**

myProxy**.**authenticate**(**"myUsername"**,** "myPassword"**);**

Console**.**WriteLine**(**"Current Balance : " **+** myProxy**.**getAccBalance**(**100**));**

Console**.**WriteLine**(**"Updating Balance by 100 "**);**

myProxy**.**updateAccBalance**(**100**,** 100**);**

Console**.**WriteLine**(**"Current Balance : " **+** myProxy**.**getAccBalance**(**100**));**

Console**.**ReadLine**();**

**}**

**}**

**}**

## Account.cs

class Account

**{**

int accNo**;**

double bal**;**

string custName**;**

string custAddress**;**

string customerType**;**

string creditRating**;**

**public** Account**(**int a**,** double b**,** string c**,** string d**,** string e**,** string f**)**

**{**

accNo **=** a**;**

bal **=** b**;**

custName **=** c**;**

custAddress **=** d**;**

customerType **=** e**;**

creditRating **=** f**;**

**}**

**public** int readAccNo**()**

**{**

**return** accNo**;**

**}**

**public** void setAccNo**(**int accNo**)**

**{**

**this.**accNo **=** accNo**;**

**}**

**public** double readBal**()**

**{**

**return** bal**;**

**}**

**public** void setBal**(**double bal**)**

**{**

**this.**bal **=** bal**;**

**}**

**public** String readCustName**()**

**{**

**return** custName**;**

**}**

**public** void setCustName**(**String custName**)**

**{**

**this.**custName **=** custName**;**

**}**

**public** String readCustAddress**()**

**{**

**return** custAddress**;**

**}**

**public** void setCustAddress**(**String custAddress**)**

**{**

**this.**custAddress **=** custAddress**;**

**}**

**public** String readCustomerType**()**

**{**

**return** customerType**;**

**}**

**public** void setCustomerType**(**String customerType**)**

**{**

**this.**customerType **=** customerType**;**

**}**

**public** String readCreditRating**()**

**{**

**return** creditRating**;**

**}**

**public** void setCreditRating**(**String creditRating**)**

**{**

**this.**creditRating **=** creditRating**;**

**}**

**}**