Implementation of Business policies using object-oriented methodologies and design patterns

Gholam Ali Shaykhian

National Aeronautics and Space Administration (NASA)
Bethune Cookman College (NAFP Fellow)
Daytona Beach, Florida

Introduction

In the early problem-solution era of software programming, functional decompositions were mainly used to design and implement software solution. In functional decompositions, functions and data are introduced as two separate entities during the design phase, and are followed as such in the implementation phase. In general, separation of function and data in a software program causes tight coupling between the two. Tight coupling means that a change in a data may require multiple changes to the design and code through the system. Also, tight coupling of data and function adversely impacts the cost of software maintenance. For example, the year 2000 phenomena (Y2K) broke many software program logics that involved date arithmetic operations (subtracting year portion of the date), processing the year portion of a date produced erroneous results. Conceptually the correction is minimal; "just change the year from two digits to four digits", however we now know that correcting the date error cost the business communities billions of dollar. The correction included fixing the software so that the year portion of a date is represented as four digits attribute versus two digits programmed earlier. The major cost was due to "tight coupling"; date was tightly coupled with all functions using the date; as such, changing the date required making several changes throughout the software systems.

The reuse of the design artifacts in functional decompositions also lacks transparency; mostly the design artifact incorporates functions needed to solve a software problem at a time. Considering that software life cycles assumed for business problems include problem analysis and design, implementation, testing and verification, deployment and maintenance phases. Where a set of robust practices required within each phase. Often practices within a phase are limited to the availability of tools, technologies and programming languages used for implementation. Software reuse in object-oriented methodologies has proven their superiority over functional decompositions. This has led to exponential growth in object-oriented market.

This paper advocates the usage of object-oriented methodologies and design patterns as the centerpieces of software solution in implementing business policies. The combine usage of object-oriented methodologies and design pattern could facilitate business

decisions and could benefit the overall software life cycle by eliminating tight coupling inherited in functional decompositions. A tutorial programming example is introduced in C++ programming language to explore the usage of object-oriented programming and design patterns. Key features of object-oriented methodologies are covered in the program example. Also, the paper examines limitations inherited in procedural programming language where function and data are two separate entities. The absent of cohesion of data and function in procedural software design exposes fundamental design deficiencies. In procedural software design, the design mandates emphasizing design solution for the problem at hand lacking generalized reuse approach.

Object-oriented methodologies

The usage of object-oriented methodology in constructing engineering and business applications has grown exponentially since the early 90's. Object-oriented software design focuses on objects versus functions and functional decompositions. An object is introduced as a distinct entity, containing its data and functions. The main features of object-oriented methodology are encapsulation, inheritance and polymorphism. Encapsulation refers to wrapping object attributes and behaviors in an enclosed entity, inheritance deals with object reuse, and polymorphism concerns with object having access to a behavior where the knowledge to the access is known at runtime.

Object

Object encapsulates the attributes (data or member data) and behaviors (function or member function) of an entity. In practice, the ill usage of objects is to transport a set of unrelated data and functions enclosed in a named entity. Using object as a transporter eliminates the benefit of object-oriented solution and would entail the same problems as known in functional decomposition. The user of the object (here after, the term client of the object will be used to refer to the term user of object) processes these unrelated data and functions and not benefiting from the strong merit of object-oriented methodologies.

Design of an object, must encompass "only" related data and functions of that object. Member data and functions of an object are tightly coupled; changes to a member data are only possible through its corresponding member functions. The client of the object is loosely coupled to the object; a client cannot directly change the object data; the request to change the object's data is sent to the object via object's member functions. A client wants to change an object data, it sends a message to the object, requesting for the change. Explicit definition of an object in this form lends itself to significant software reuse.

In object-oriented design and programming, object data and member function supporting object are encapsulated as one entity known as a user-define class data type. A class wraps general characteristic of an object, specific object of the class are defined as needed. Listing-1 shows a Person class, the Person class wraps the general characteristics for a person. Class provides a mechanism to hide its members from public access through private and protected sections. In this example access to both name and identification is limited to internal class Person, represented by open and close curly

brackets. Client of class Person may request the class private attributes through accessor member functions. The protected section of Person class is accessible to other classes which derive from Person class, the example later shows that BankEmployee and BankCustomer classes both derive from class Person and will have access to the protected section of class Person.

```
// Person.h <Header file>
#ifndef PERSON_H   // multiple definition guard
#define PERSON_H
#include <iostream>  // for input/output stream
#include <string>       // for string
using namespace std;
// Person class serves as a base class for Customer and Employee classes. The object
// creation of this class is limited to BankFactory class. The constructor is placed in
// protected section of the class to disallow public access.
class BankFactory;     //forward declaration to allow dependency
class Person {
friend class BankFactory;      // Dependency statement - friend class
friend ostream& operator <<(ostream&, const class Person&); // output
public:
   virtual ~Person() {}                  // runtime binding-avoid memory leak
   string getName();                     // accessor member function
   string getIdentification();           // accessor member function
   void setName(string&);                // mutator member function
   void setIdentification(string&);      // mutator member function
   virtual void show();                  // polymorphic member function
protected:
   Person() {}                           // disallow public object creation
private:
   string name;
   string identification;
};
#endif

// Person.cpp <implementation file>
#include "Person.h"
ostream& operator <<(ostream & output , const class Person & P) {
   output << P.identification << "\t"<< P.name << "\n";
   return output; // accommodate cascading the << operator
}
string Person::getName() {   return this->name; }
string Person::getIdentification() {   return this->identification;}
void Person::setName(string &N) {   this->name = N; }
void Person::setIdentification(string &Id) {  this->identification = Id;}
void Person::show() {  cout << *this; }
```

Listing-1 Description of a class Person

Object relationships

Relationships among objects are similar to those known to us in real life. Let suppose an object of type Person has an account with a bank. A set of attributes and behavior describe the Person object. A bank teller processes the person's bank account object. The bank teller is an employee of the bank. A bank itself is branch in a bank chain. A person object, bank teller object, and a bank itself have roles and responsibilities in processing a bank account. Object relationships formalizes the relationships among objects, these relationships are commonly known as *knows-a*, *is-a*, *has-a* and *depends-a*. Additional relationships among objects can be derived from these relationships, for example *with-a*. *Knows-A* relationship describes an association between two objects; for example, a bank teller knows a bank customer through the customer's bank account information. This knowledge can be unidirectional (a bank customer knows a bank teller) or bi-directional (both the bank teller and the bank customer know each others).

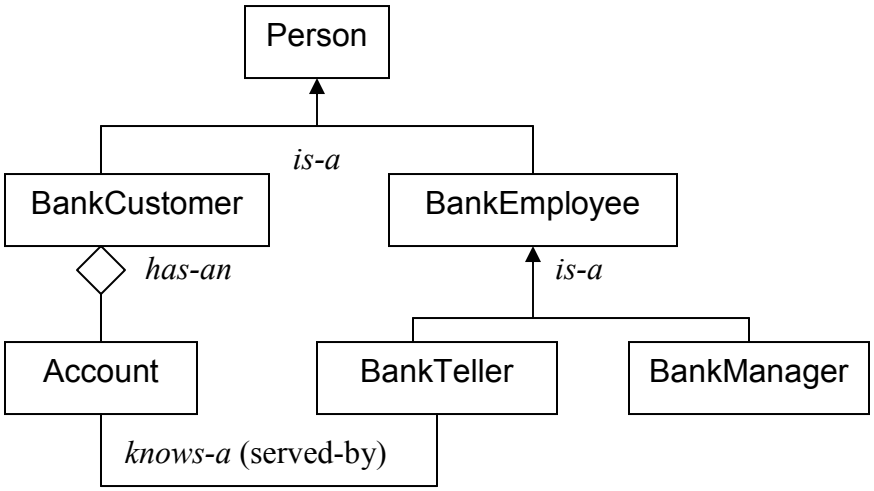

Figure-1 Object relationships

The bank teller, manager and customer are objects of type Person; we write by describing a bank teller is an employee, an employee is a person (*is-a* relationship) and a bank customer is a person. *Is-a* relationship forms strong relation among objects, known as inheritance. Inheritance among objects introduces two related topic of equal important (1) object access and (2) object ownerships. Object access deals with object having access to its parent or own class members. In Figure-1, the objects of class Person (hereafter we refer to Person as a base class, and objects of Person as base objects, super objects, or parent objects and we refer to the BankCustomer, BankTeller and BankCustomer class as derived class where the objects of derived class is derived objects, sub-objects, or child objects) has access to public members of class Person. The derived objects have access to all the public members of derived class plus all public members of its corresponding base

class. Memory allocate for an object is expressed here as object ownership. A base object owns its class data and a derived object owns its class data plus the class data of its corresponding base class.

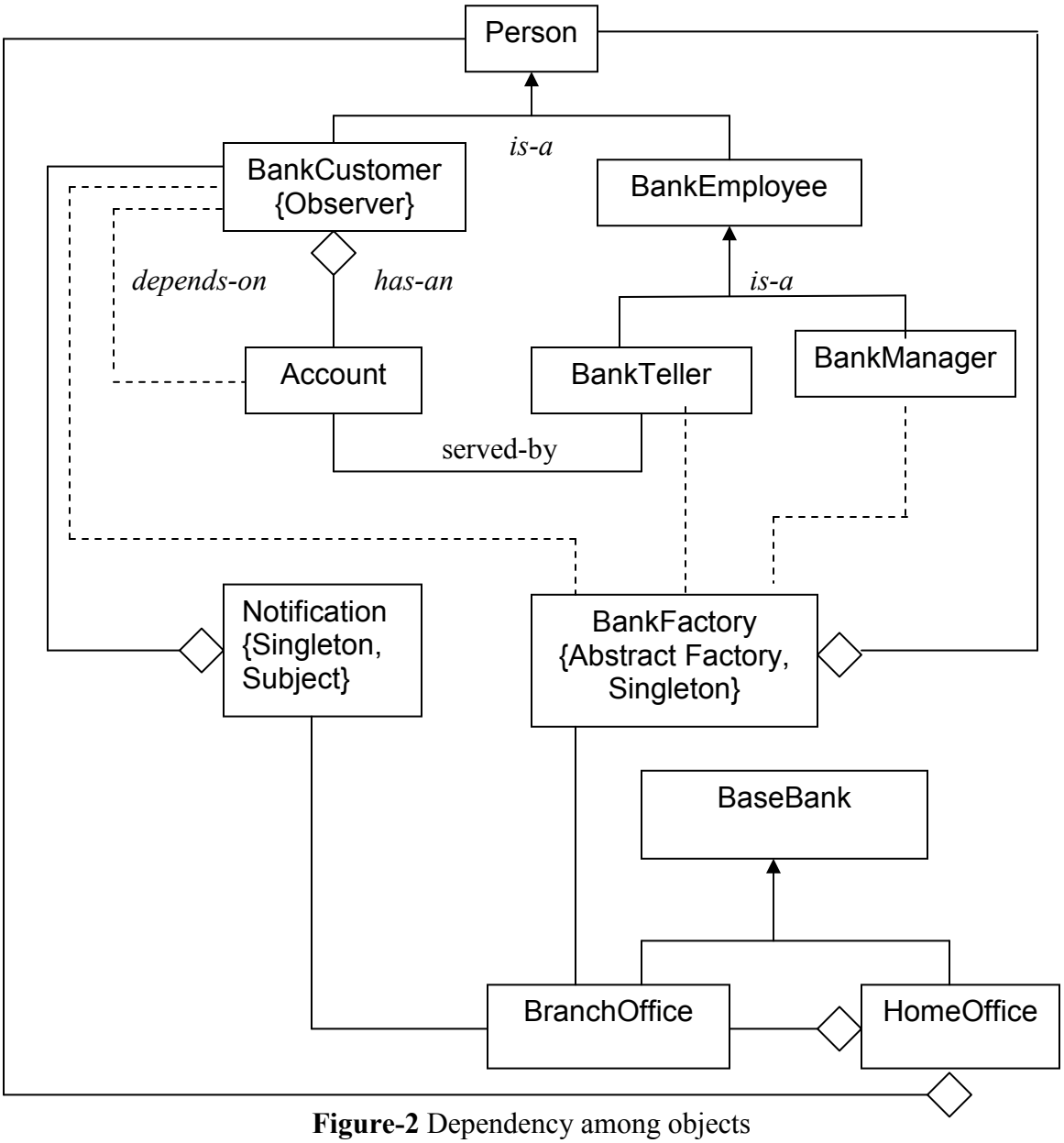

Figure-2 Dependency among objects

Has-a relationship is when an object is composed of other objects. A customer has accounts, bank has employees, are examples of has-a relationship.

The need to establish relationships among objects that deal with limited privileges may arise in business problems. Limited privileges can be modeled as *depends-a* relationships in object-oriented methodologies. For example, a bank may want to restrict creation of a new account. The restriction might be to allow certain employee to have privilege to open new customer account. In this situation, object dependency can be used to regulate such restrictions among objects. In Figure-2, dash line between objects represents dependency. For example, in figure, dependency is established between BankFactory and BankCustomer classes, these two classes are connected with dash line modeling their dependency.

Polymorphism

Member functions of a class are accessible by their corresponding objects or object pointers. In object-oriented programming, an object pointer of a derived class can be assigned onto an object pointer of its base class. Assigning derived object pointer onto base pointer enhances programming. The enhancement would be to declare a single base pointer and use it with its derived objects during runtime. The assignment of derive object pointer onto base object pointer is permitted since a derived object also contain its base sub-object portion. However, declaring a base pointer binds to its members at compile time (static binding) causing runtime misalignment. The misalignment is, if a base pointer is pointing to a derived object then calling its derived members should yield to a call to derived member; but it does not, it always yields to a call to the base members. It remains that a base pointer binds to its non-virtual base members at compile time and it binds to its virtual members at runtime. A base pointer is used to call members of its derived classes through polymorphism.

Polymorphism enables object pointer to bind to its virtual member functions at run time (late binding). The procedure is as follow:

- The target member functions must be declared as virtual in base class.
- Each derived class provides its own specific implementations of the virtual member functions.
- The derived class must use the same interface for its virtual member.
- The internal implementation of the derived versions varies from its base class.
- A base pointer points to a derived class.
- Then a call to a derived virtual member would yield to derived member.

The term polymorphism refers to many form or shapes. Since each derived class use the same virtual member functions interface and change the internal implementation of the functions, the term polymorphism is used.

Listing-1 shows the person class with a virtual member function, show(), later, the BankCustomer (Listing-2) class defines its own specific show() member function. Since the show() member function is declared as virtual, the base pointer delays binding to it until runtime. For example, if a base pointer is pointing to BankCustomer object then a call to show() would result BankCustomer::show(). What is described here is known as polymorphism.

```
// BankCustomer.h <Header file>
```

```
#ifndef BankCustomer_H      // multiple definition guard
#define BankCustomer_H
#include "Person.h"    // for the base class
#include <map>         // for BankCustomer's account
// BankCustomer class is a derived class of Person class. The object creation of this class
// is limited to BankFactory class. The constructor is placed in protected section of the
// class to disallow public access.
class BankFactory;     // forward declaration to allow dependency
class Account;         // forward declaration to allow compilation
class BankCustomer : public Person {
friend class BankFactory; // Dependency statement –to have access to the constructors.
public:
  virtual ~BankCustomer();   // virtual destructor
  virtual void show();       // object binds to this member function at runtime.
  Account* createAccountObject(double);
  Account* getAccountObject(string&);
  Person* getServedByObject(string&);
  void openAccount(string, double);
protected:
  BankCustomer() {}
private:
  map<string,Account*> accounts;   // BankCustomers, multiple accounts
};
#endif BankCustomer_H

// BankCustomer.cpp <implementation file>
#include "BankCustomer.h"
#include "Account.h"
#include "Notification.h"
Account* BankCustomer::createAccountObject(double d) {
   return new Account(d); // create an Account() object
}
Account* BankCustomer::getAccountObject(string &s) {
  map<string,Account*>::iterator i;
  i = accounts.find(s);
  if (i!= accounts.end())
        return (*i).second;
  else
        return 0;
}
Person* BankCustomer::getServedByObject(string &s) {
  map<string,Account*>::iterator i;
  i = accounts.find(s);
  if (i!= accounts.end())
        return (*i).second->getServedBy();
  else
```

```
        return 0;
}
void BankCustomer::show() {
  Person::show();
  map<string,Account*>::iterator i;
  for (i=accounts.begin(); i != accounts.end(); i++) {
        cout << "\t" << i->first << "\t";
        cout << (*i).second->getBalance() << "\n";
  }
}
void BankCustomer::openAccount(string s, double amount) {
  Account *K = this->createAccountObject(amount);
  pair<map<string,Account*>::iterator,bool> r;
  r = accounts.insert(make_pair(s,K));
  if (!r.second) delete K;                  // prevent memory leak
}
BankCustomer::~BankCustomer() {
  map<string,Account*>::iterator i;
  for (i=accounts.begin(); i != accounts.end(); i++) {
        delete (*i).second;      //avoid memory leak!
  }
  Notification *N=Notification::instance();
  N->unregisterBankCustomer(this);
}
```

Listing-2 Description of class BankCustomer

Working with the bank example

A C++ programming example provided to demonstrate the use of object-oriented and design pattern to implement business policy. A business practices is governed by a set of rules and regulations, the rules and regulations of a business are referred in this paper as "business policies". Conducting a business policy through manual procedure may result in inconsistencies and could require significant resources. Alternatively, providing software solution to automate business policies may prove beneficial. Software solutions bear initial development cost, and thereafter maintenance cost. Reduction of software maintenance cost can be achieved through robust design and implementation. For the reasons of couplings and lack of cohesions stated earlier, object-oriented solutions are preferred to functional solutions. The preferences of object-oriented are described below:

- Object data and member functions are encapsulated as one entity.
- Object data are hidden from the client.
- The accesses to object data are limited to its member functions.
- A client needing object data makes request through a public member functions.
- Changes to object data are centralized within the object.

The usage of object-oriented methodologies and design patterns as the centerpieces of software solution in implementing business policies is advocated throughout this paper.

The program example incorporates a set of fictitious business policies as a means to advocate object-oriented solutions. Listed below are:

1. Only home office branch has the authority to open new customer accounts, all branches are allowed to assist existing customers.
2. Different account type (saving, checking accounts) is created for a valid customer.
3. The customers will receive notifications when account balance fall below zero.
4. Customer's name and identifications are required when the home branch opens a new account.

Items 1 through 4 described above, are sample examples of "business policy". Figure-2 shows the relationships among objects, these objects are used to explain these items.

1. Only home office branch has the authority to open new customer accounts, all branches are allowed to assist existing customers.

In a large software system, there can be thousands of objects developed by a team of hundred developers. Often visual inspections, or manual tracking of requirement is a major undertaking. From the implementation points of view, functional programming languages provided no automated means to regulate this particular requirement. So we had no choice in the matter other than resorting to visual or manual inspections. Object-oriented methodologies and design patterns include capabilities that make the automation of these sorts of requirements relatively seamless. To achieve this, the design includes the following tasks:

- Write the BankCustomer constructors (shown in Listing-2) in the protected section of its class. This will disallow the creation (definition) of bank customer objects.
- Grant explicit permission to HomeBranch object to have access to the protected section of BankCustomer class. Writing friend class HomeBranch can do this.

The friendship between two classes establishes object dependency. The dependency between these two objects is by granting exclusive access to home branch object so that it can access the customer constructors written in protected sections.

The above scenario is discussed without making use of a design patterns. Subsequently the abstract factory and singleton design patterns (introduced by Gamma, Helm, Johnson and Vlissides) are used in the example program (Shown in Listing-3) to accommodate future changes, hence minimizing the cost of software maintenance.

```
// BankFactory.h <Header file>
#ifndef BANKFACTORY_H
#define BANKFACTORY_H
#include "Person.h"
// The BankFactory class is an abstract factory and singleton class. The class defines a
// container for all Person. The create member functions use a base pointer of Person
// class. The base pointer can point to a base object or any of its derived objects.
class BankFactory {
public:
  static BankFactory * instance();     // returns singleton object.
  Person * createCustomer();
  Person * createManager();
  Person * createTeller();
```

```
private:
        // additional attributes for BankFactory
};
#endif BANKFACTORY_H

// BankFactory.cpp <implementation file>
#include "BankFactory.h"
#include "BankCustomer.h"
#include "BankTeller.h"
#include "BankManager.h"
BankFactory* BankFactory::instance() {
  static BankFactory theInstance;  // Singelton object
  return &theInstance;
}
Person * BankFactory::createCustomer() {  return new BankCustomer(); }
Person * BankFactory::createManager() {  return new BankManager();   }
Person * BankFactory::createTeller() {  return new BankTeller(); }
```

Listing-3 Description of BankFactory class

2. Different account type (saving, checking accounts) is created for a valid customer. Writing a friend statement in Account class, and placing the Account constructor in private section of its class accomplishes this requirement. Listing-4 shows the Account class.

```
// Account.h <Header file>
#ifndef ACCOUNT_H
#define ACCOUNT_H
// The Account object is created through the Customer object.
#include <iostream>          // for input/output stream
using namespace std;
class Person;                // forward declaration
class Account
{
  friend class BankCustomer; // Exclusive privilege to BankCustomer class
  friend ostream& operator <<(ostream&, const class Account&);
public:
  void deposite(double,Person*);
  double withdraw(double,Person*);
  double getBalance();
  Person* getServedBy();
protected:
  Account(double=0);
  ~Account() {}
private:
  double balance;
  Person *lastServedBy;
```

```
};
#endif ACCOUNT_H

// Account.cpp <implementation file>
#include "Account.h"
ostream& operator <<(ostream & output , const class Account & A) {
  cout << "\nOutput balance = " << A.balance<< "\n";
  return output; // return for cascading the << operator
}
void Account::deposite(double b, Person *servedBy) {
  balance +=b;
  lastServedBy = servedBy;
}
double Account::withdraw(double w, Person *servedBy) {
  balance -=w;
  lastServedBy = servedBy;
  return balance;
}
double Account::getBalance() { return balance; }
Person* Account::getServedBy() {return lastServedBy; }
Account::Account(double b): balance(b),lastServedBy(0) {}
```

Listing-4 Description of Account class

3. The customers will receive notifications when account balance fall below zero.

Subject-Observers design pattern is used to implement this requirement. Customers object register its interest with the subject object (Notification object). When the account balance falls below zero, the subject object sends notifications to the affected observer. Notification class provides a registerCustomer() method to record the observer's interest, and a notifyCustomer() method to notify bank customer when account balance falls below zero listed in Listing-5.

```
// Notification.cpp <Header file>
#ifndef Notification_H
#define Notification_H
#include <list>
#include "Person.h"
// This class defines a static container for all customers who will receive notifications
// when the withdrawal amount exceeds available balance in their account.
class Notification {
public:
  void registerCustomer(Person*);
  void unregisterBankCustomer(Person*);
  static Notification * instance();  // return the instance of the singleton object
  void notifyCustomer(Person*,string,double,double);
  ~Notification() {}     // destructor
```

```
protected:                // disallow public access
  Notification() {}
private:
  static std::list<Person*> customers; // customers who receive notifications from Bank
};
#endif Notification_H

// Notification.cpp <implementation file>
#include "Notification.h"
#include "BankCustomer.h"
std::list<Person*> Notification::customers;
Notification * Notification::instance() {
  static Notification N;
  return &N;   // return singleton object
}
void Notification::notifyCustomer(Person *bankCustomer,string accountType,
                    double amountAvailable, double amountWithdrawal) {
   string message = "Non-Sufficent Funds <" + accountType+"> ";
   BankCustomer *C = (BankCustomer*)bankCustomer;
   Person *accountServedBy = C->BankCustomer::getServedByObject(accountType);
   cout <<*C<<"\tAmount Available: "  << amountAvailable;
   cout <<"\t\tAmount Withdrawn: "  << amountWithdrawal <<"\n";
   cout <<"\t"<<message << "\n";
   cout <<"\tNotification issued by Bank Employee: "<<accountServedBy->getName();
   cout << "\n\n";
}
void Notification::registerCustomer(Person *C) { customers.push_back(C); }
void Notification::unregisterBankCustomer(Person  *C) {
   std::list<Person*>::iterator i=customers.begin();
   bool deleted=false;
   while (!deleted && i!=customers.end())
       if ((*i) == C)       {
           deleted = true;
           customers.erase(i);
       }
       else i++;
}
```

Listing-5 Description of Notification class

4. Customer's name and identifications are required when the home branch opens a new account.

BankCustomer provides a constructor with two string parameters (Listing-2). This constructor is used to create a customer object with name and identification.

Summary

In this paper, an introduction to object-oriented methodologies and design patterns are presented. The benefits of combine usage of object-oriented methodologies and design patterns to facilitate business decisions are investigated. Eliminating tight coupling inherited in functional decompositions and establishing cohesion within an object are emphasized strongly.

Acknowledgments

Special thanks to my wife Linda Shaykhian for her inspiration and testing example in the development of the C++ Bank program listed in appendix section of this paper.

Bibliography

1. Gamma, A. Helm, R. Johnson, R. Vlissides, J. "Design Patterns, Elements of Reusable Object-Oriented Software," New York: Addison-Wesley, 1995.
2. Stroustrup, B. "The C++ Programming Language," New York: Addison-Wesley, 2000.

GHOLAM ALI SHAYKHIAN

Gholam Ali Shaykhian is a software engineer with National Aeronautics and Space Administration (NASA), Kennedy Space Center (KSC), Shuttle Processing Directorate. He is serving as a visiting Instructor of Computer Science at Bethune Cookman College in Daytona Beach, Florida under the National Administrator Fellowship Program (NAFP). Ali has received a Master of Science (M.S.) degree in Computer Systems from University of Central Florida in 1985 and a second M.S. degree in Operations Research from the same university in 1997. His research interests include Object-Oriented methodologies and design patterns. He has taught information system and computer science courses for Bethune Cookman College, Webster University, Barry University and University of Central Florida. Mr. Shaykhian is a senior member of Institute of Electrical and Electronics Engineering (IEEE) and is Vice-Chair (2005) and Education Chair (2003-2005) of IEEE Canaveral section.

Appendix

```
// BankEmployee.h <Header file>
#ifndef BankEmployee_H     // multiple defintion guard
#define BankEmployee_H
#include "Person.h"    // base class
// BankEmployee class encapsulates BankEmployee data. The object creation of this class
// is limited to the BankFactory class. The constructor is placed in protected section of the
// class to disallow public access.
```

```
class BankFactory;              //forward declaration to allow dependency
class BankEmployee : public Person {

friend class BankFactory;      // Dependency statement -to have access to constructor
public:
  ~BankEmployee();   //  polymorphic  destructor
   virtual void show();          // object binds to this member function at runtime.
protected:
   BankEmployee(const string &, const string &);    // disallow public access
private:
   // additional attributes for BankEmployee
};
#endif

// BankEmployee.cpp <implementation file>
#include "BankEmployee.h"
void BankEmployee::show() { Person::show();}
BankEmployee::~BankEmployee() {}
BankEmployee::BankEmployee(const string &N, const string &D) : Person(N,D) {}
```

Listing-6 Description of BankEmployee class

```
// BankTeller.h <Header file>
#ifndef BankTeller_H// multiple definition guard
#define BankTeller_H
#include "BankEmployee.h"
//  BankTeller class inherits from the Employee class. The object creation of this class is
//  limited to BankFactory class.
class BankFactory;     //forward declaration to allow dependency
class BankTeller : public BankEmployee {
friend class BankFactory;      // Dependency statement - friend class
public:
  ~BankTeller();         // destructor
  virtual void show(); // polymorphic member function
protected:
  BankTeller(const string &, const string &); // disallow public access
private:
  // additional attributes for BankTeller
};
#endif

//BankTeller.cpp <implementation file>
#include "BankTeller.h"
void BankTeller::show() {  BankEmployee::show();}
BankTeller::~BankTeller() { }
BankTeller::BankTeller(const string &N, const string &D): BankEmployee(N,D){}
```

Listing-7 Description of BankTeller class

```
// BankManager.h <Header file>
#ifndef BankManager_H                // multiple definition guard
#define BankManager_H
#include "BankEmployee.h"
// BankManager class inherits from the Employee class. The object creation of this class
// is limited to BankFactory class.
class BankFactory;     //forward declaration to allow dependency
class BankManager : public BankEmployee {
friend class BankFactory;      // Dependency statement - friend class
public:
  ~BankManager();     // polymorphic destructor
  virtual void show(); // object binds to this member function at runtime.
protected:
  BankManager(const string &, const string &);// construct object
private:
  // additional attributes for BankManager
};
#endif

// BankManager.cpp <implementation file>
#include "BankManager.h"
void BankManager::show() {  BankEmployee::show(); }
BankManager::~BankManager()  {}
BankManager::BankManager(const string &N,const string &D): BankEmployee(N,D){}
```

Listing-8 Description of BankManager class

```
// BaseBank.h <Header file>
#ifndef BASEBANK_H        // multiple defintion guard
#define BASEBANK_H
#include <iostream>             // for input/output stream
#include <string>               // for string
using namespace std;
// BaseBank encapsulates the base data for the banks.
class BaseBank {
friend ostream& operator <<(ostream&, const class BaseBank&); // output
public:
   virtual ~BaseBank(){}                // destructor
   string getBranchName();              // accessor member function
   string getBranchAddress();           // accessor member function
   void setBranchName(string&);         // mutator member function
   void setBranchAddress(string&);    // mutator member function
   virtual void show();           // polymorphic member function.
   BaseBank(const string &, const string &);// construct object
private:
   string branchName;
   string address;
```

```
};
#endif BASEBANK_H

// BaseBank.cpp <Implementation file>
#include "BaseBank.h"
ostream& operator <<(ostream &output, const class BaseBank &B) {
   cout << "\nBank: " << B.branchName << "\t" << B.address << "\n";
   return output; // return for cascading the << operator
}
string BaseBank::getBranchName() {   return branchName; }
string BaseBank::getBranchAddress() {  return address; }
void BaseBank::setBranchName(string &N) {branchName = N; }
void BaseBank::setBranchAddress(string &addr) {address = addr; }
void BaseBank::show() { cout << *this; }
BaseBank::BaseBank(const string &N, const string &a):branchName(N), address(a) {}
```

Listing-9 Description of BaseBank class

```
// BranchOffice.h <Header file>
#ifndef BranchOffice_H
#define BranchOffice_H
#include <list>
#include "BaseBank.h"
#include "BankCustomer.h"
// The BranchOffice class encapsulates a bank branch data. This class defines a container
// for customer object pointer that belong to a branch.
class BranchOffice :public BaseBank {
public:
  ~BranchOffice() {}     // destructor
  virtual void show();  // objects bind to this member function at runtime.
//  The addxxxxx member functions serve as utility member functions to add a customer
// object in class container.
  void addCustomer(Person*);
  void addTeller(Person*);
  void addManager(Person*);
  double getRandomAmount();          // produce a random number
// The openBank() member function is utilized to call the openAccount() member
// function to create customer's account. The customer's account is populated with
// simulated data.
  void openBank();
  void openAccount(BankCustomer *);
//  The processCustomerAccount() utilizes the existing customers accounts to withdraw a
//  random amount and issue notifications when account amount becomes negative.
  void processCustomerAccounts();
  BranchOffice(const string &, const string &);         // constructor
private:
```

```
  std::list<Person*> customers;   // list<> container for all customers of a Branch
  std::list<Person*> employees;  // list<> container for all Tellers/Manager of a branch
};
#endif BranchOffice_H

//  BranchOffice.cpp <implementation file>
#include "BranchOffice.h"
#include "BankFactory.h"
#include "Notification.h"
#include "BankCustomer.h"
#include "Account.h"
#include <stdlib.h>
#include <time.h>
void BranchOffice::show() {
   BaseBank::show();
   list<Person*>::iterator i;
   cout << "\nEmployee Members:\n"<< "_____\n";
   for (i=employees.begin(); i != employees.end(); i++)
         (*i)->show();
   cout << "\nCustomer Members:\n"<< "_____\n";
   for (i=customers.begin(); i != customers.end(); i++)
         (*i)->show();
}
void BranchOffice::addCustomer(Person *C) { customers.push_back(C); }
void BranchOffice::addTeller(Person *T) {  employees.push_back(T); }
void BranchOffice::addManager(Person *M) { employees.push_back(M); }
void BranchOffice::openAccount(BankCustomer *C) {
   C->openAccount(string("Checking Account"),getRandomAmount());
   C->openAccount(string("Savings  Account"),getRandomAmount());
   Notification *N=Notification::instance();
   N->registerCustomer(C);
}
void BranchOffice::openBank() {
   std::list<Person*>::iterator i;
   for(i=customers.begin(); i!=customers.end(); i++)
         BranchOffice::openAccount((BankCustomer*)(*i));
}
double BranchOffice::getRandomAmount() {
   srand((unsigned) time( NULL ));    // wait 1 second
   clock_t goal;
   goal = (clock_t)3 * CLOCKS_PER_SEC + clock();
   while( goal > clock() );
   srand((unsigned) goal);
   return double(rand());
}
void BranchOffice::processCustomerAccounts() {
```

```
  Account       *SavingAccount;
  Account       *CheckingAccount;
  BankCustomer        *C;
  double        amount;
  double        amountWithdrawal;
  double        amountAvailable;
  Notification * N=Notification::instance();
  std::list<Person*>::iterator i;
  std::list<Person*>::iterator j=employees.begin();
  for(i=customers.begin(); i!=customers.end(); i++) {
    C = (BankCustomer*)(*i);
    SavingAccount   = C->getAccountObject(string("Savings  Account"));
    CheckingAccount = C->getAccountObject(string("Checking Account"));
    amountAvailable  = SavingAccount->getBalance();
    amountWithdrawal = getRandomAmount()/2;
    amount = SavingAccount->withdraw(amountWithdrawal,(*j));
    if (amount < 0)       // send Notification to customers
      N->notifyCustomer(C,"Savings  Account",amountAvailable,amountWithdrawal);
   amountAvailable  = CheckingAccount->getBalance();
   amountWithdrawal = getRandomAmount()/2;
   amount = CheckingAccount->withdraw(amountWithdrawal,(*j));
   if (amount<0)         // send Notification to customers
     N->notifyCustomer(C,"Checking Account",amountAvailable,amountWithdrawal);
   j++;
  if (j == employees.end()) j =employees.begin(); // rotate among employees
   }
}
BranchOffice::BranchOffice(const string &N, const string &addr): BaseBank(N,addr) {}
```

Listing-10 Description of BranchOffice class

```
// HomeOffice.h <Header file>
#ifndef HomeOffice_H
#define HomeOffice_H
#include <map>
#include <list>
#include "BaseBank.h"
#include "Person.h"
#include "BranchOffice.h"
// The HomeOffice class defines three static contains for the concrete objects of the bank
// branch data, the customers data and the employees data. This class inherits from the
// BaseBank.
class HomeOffice :public BaseBank {
public:
  ~HomeOffice(){ }              // destructor
  virtual void show();          // polymorphic member function
```

```
//  The openBank() member function is utilized to populated the bank branch, with
//  customers and employees with simulated data.
   void openBank();
// The addxxxxx member functions serve as utility member functions to create and return
//  an object. The object creation of customers, tellers and managers is limited to the
//  BankFactory class.
   Person* addCustomer(string &S1, string &S2);
   Person* addTeller(string &S1, string &S2);
   Person* addManager(string &S1, string &S2);
// The populateBankData() uses the assignxxxx utility functions to assign bank
// employees and customers to their respected branch.
   BranchOffice* getBranchObject(int);
   void assignTellers(BranchOffice *B);
   void assignManagers(BranchOffice *B);
   void assignCustomers(BranchOffice *B);
   void populateBankData();
// The processBankAccounts() simulates the customer bank data by processing accounts
// with a random value. The subject-observer design patterns is utilized with this member
// function.
   void processBankAccounts();
// The closeBank() member function is utilized to clean up memory allocated for the bank
// branch, customers and employees to avoid memory leak.
   void closeBank();
   HomeOffice(const string &, const string &);  // construct object
private:
   // multimap<branch name, customer object> container for all customers
   static std::multimap<string,Person*> customers;
   static std::multimap<string,Person*> employees;   // multimap<> for all employees
   static std::list<BranchOffice*> banks; // list<> container for all bank branches
};
#endif HomeOffice_H

// HomeOffice.cpp <implementation file>
#include "HomeOffice.h"
#include "BankFactory.h"
#include "Person.h"
#include "BankTeller.h"
#include "BankManager.h"
#include "BankCustomer.h"
std::multimap<string,Person*> HomeOffice::customers; // static definition
std::multimap<string,Person*> HomeOffice::employees; // static definition
std::list<BranchOffice*> HomeOffice::banks; // static definition
void HomeOffice::show() {
   BaseBank::show();  // show the Home Office
   list<BranchOffice*>::iterator i;
   for (i=banks.begin(); i != banks.end(); i++) {
```

```
      (*i)->BranchOffice::show();
      cout <<"\n\n";
   }
   cout <<"\n\n";
}
Person* HomeOffice::addCustomer(string &S1, string &S2) {
   BankFactory *BF =  BankFactory::instance();
   Person *P1= BF->BankFactory::createCustomer();
   P1->setName(S1);
   P1->setIdentification(S2);
   return P1;
}
Person* HomeOffice::addTeller(string &S1, string &S2) {
   BankFactory *BF =  BankFactory::instance();
   Person *P1= BF->BankFactory::createTeller();
   P1->setName(S1);
   P1->setIdentification(S2);
   return P1;
}
Person* HomeOffice::addManager(string &S1, string &S2) {
   BankFactory *BF =  BankFactory::instance();
   Person *P1= BF->BankFactory::createManager();
   P1->setName(S1);
   P1->setIdentification(S2);
   return P1;
}
BranchOffice* HomeOffice::getBranchObject(int k) {
   std::list<BranchOffice*>::iterator i=banks.begin();
   if (1!=k)  i++;
   return (*i);
}
void HomeOffice::populateBankData() {
  // Lets have a few customers for the first Branch
  BranchOffice *BO = getBranchObject(1);
  string S1 = BO->getBranchName();
  customers.insert(make_pair(S1,addCustomer(string("Fluffy Tweek"),string("C111"))));
  customers.insert(make_pair(S1,addCustomer(string("Toots Carver"),string("C222"))));
  customers.insert(make_pair(S1,addCustomer(string("Otis Emilliom"),string("C333"))));
  customers.insert(make_pair(S1,addCustomer(string("Coco Shagans"),string("C444"))));
 // Lets have Bank Tellers
  employees.insert(make_pair(S1,addTeller(string("Buco Calais"),string("T111"))));
  employees.insert(make_pair(S1,addTeller(string("Seih Fox"),string("T222"))));
  // The Bank Manager
  employees.insert(make_pair(S1,addManager(string("Chico Ham"),string("M111"))));
  // Lets have a few customers for the second Branck
  BO = getBranchObject(2);
```

```
  S1 = BO->getBranchName();
  customers.insert(make_pair(S1,addCustomer(string("Blondie Shoe"),string("C555"))));
  customers.insert(make_pair(S1,addCustomer(string("Moe Howard"),string("C666"))));
  customers.insert(make_pair(S1,addCustomer(string("Curly Stooge"),string("C777"))));
  customers.insert(make_pair(S1,addCustomer(string("Larry Howard"),string("C888"))));
  // Lets have Bank Tellers
  employees.insert(make_pair(S1,addTeller(string("Kahlua King"),string("T333"))));
  employees.insert(make_pair(S1,addTeller(string("Josie Eyster"),string("T444"))));
  // The Bank Manager
  employees.insert(make_pair(S1,addManager(string("Zeke Beach"),string("M222"))));
}
void HomeOffice::openBank() {
    // Lets add a few banks
    banks.push_back(new BranchOffice(string("Mistletoe Branch"),string("B111")));
    banks.push_back(new BranchOffice(string("Holly Branch"),string("B222")));
    // Lets add a few customers, tellers and manager to each bank
    HomeOffice::populateBankData();
    // Assign Tellers, Managers, and Customers to each branch
    std::list<BranchOffice*>::iterator i=banks.begin();
    for (i=banks.begin();i !=banks.end(); i++) {
          HomeOffice::assignTellers((*i));
          HomeOffice::assignManagers((*i));
          HomeOffice::assignCustomers((*i));
    }
    // Continue operation within each branch
    for (i=banks.begin(); i!=banks.end(); i++)
          (*i)->openBank();
}
void HomeOffice::processBankAccounts() {
   std::list<BranchOffice*>::iterator i=banks.begin();
    for (i=banks.begin(); i!=banks.end(); i++)
          (*i)->BranchOffice::processCustomerAccounts();
}
void HomeOffice::closeBank() {
   std::multimap<string,Person*>::iterator i;
   for (i=customers.begin(); i != customers.end(); i++)
          delete (*i).second;
   for (i=employees.begin(); i != employees.end(); i++)
          delete (*i).second;
   std::list<BranchOffice*>::iterator j;
   for (j=banks.begin(); j != banks.end(); j++)
          delete (*j);
}
void HomeOffice::assignTellers(BranchOffice *B) {
   string S = B->getBranchName();
   BankTeller *T;
```

```
  Person *P;
  std::multimap<string,Person*>::iterator i=employees.begin();
  while (i != employees.end()) {
        P = (*i).second;
        if ((*i).first == S) {
              T=dynamic_cast<BankTeller*>(P);
              if (T != NULL) B->BranchOffice::addTeller(P);
        }
        i++;
  }
}
void HomeOffice::assignManagers(BranchOffice *B) {
  string S = B->getBranchName();
  BankManager *M;
  Person *P;
  std::multimap<string,Person*>::iterator i=employees.begin();
  while (i != employees.end()) {
        P = (*i).second;
        if ((*i).first == S) {
              M=dynamic_cast<BankManager*>(P);
              if (M != NULL) B->BranchOffice::addManager(P);
        }
        i++;
  }
}
void HomeOffice::assignCustomers(BranchOffice *B) {
  string S = B->getBranchName();
  BankCustomer *C;
  Person *P;
  std::multimap<string,Person*>::iterator i=customers.begin();
  while (i != customers.end()) {
        P = (*i).second;
        if ((*i).first == S) {
              C=dynamic_cast<BankCustomer*>(P);
              if (C != NULL) B->BranchOffice::addCustomer(P);
        }
        i++;
  }
}
HomeOffice::HomeOffice(const string &N, const string &addr) : BaseBank(N,addr){}
```

Listing-11 Description of HomeOffice class