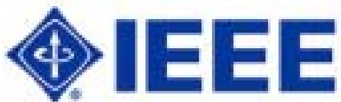
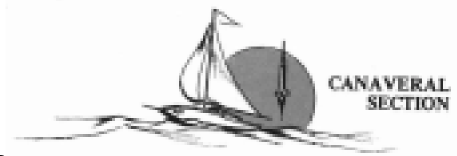




IEEE Canaveral Section Newsletter

Vol. VI No. 1 http://www.ieee.org/canaveral January – April 2005

A New Year for the Canaveral Section

Since this is one of my first opportunities to address the Canaveral Section of the IEEE, I'd like to introduce myself. My name is Chuck Chapman and I have the wonderful opportunity to serve as chairman of the Canaveral Section this year. I moved to the Space Coast in 1998 after working for the Georgia Tech Research Institute the previous 13 years. Initially I worked for Dynacs Engineering Company on the Engineering Development Contract at the Kennedy Space Center. In 2000 I accepted a position with NASA to work on the ill-fated Checkout and Launch Control System (CLCS). Since the cancellation of CLCS I've worked for the Shuttle Processing directorate at KSC, primarily as a software developer and cochairman of the Technical Review Panel on the PCGOAL2 project. I am also a licensed Extra class amateur radio operator (since 1971) with the call sign WB4UIH.

Shortly after moving to the Space Coast, I started attending the monthly meetings of the Canaveral Section. Past chair Kathy Rinehart helped correct a long-standing error in my membership status – for years I had been listed as an associate member with IEEE headquarters when there was no reason I should not be a full member (Note: I urge all members in the Canaveral Section to also check their membership status, which is listed on your IEEE membership card. Let me know if your status is not correct). In later years, Kathy was very helpful in upgrading my member status to Senior Member of the IEEE.

Before moving to the Space Coast, I'd never been active in the IEEE, although I'd been a member since 1977. I didn't know what I'd been missing! The section meetings I've attended since moving here have all been very interesting and informative, and participating as an officer in the section has been very rewarding. I urge all members of the Canaveral Section to attend as many meetings as possible this year, if for nothing else but the free pizza! (at most meetings) We'll try to have meeting topics on a variety of topics, and I'm definitely open to suggestions on topics for future meetings. Send me your ideas! Also, the Canaveral Section has traditionally held their meetings on Thursday evenings. If you think another time might be more convenient for a lot of our members to attend meetings, let me know. You can email me at ccmail@bellsouth.net or sec.canaveral@ieee.org.

With your help and participation, we can make this another successful year for the Canaveral Section!

What's Inside…..

A New Year for the Canaveral Section 1
Adam (Buddy) Kissiah Gives Fascinating Lecture on Cochlear Implant Invention 2
Joint Canaveral/Melbourne Life Member Chapter Holds First Meeting 2
NASA Engineer Ali Shaykhian Lectures on Design Patterns 3
Canaveral Section To Offer Seminar on C++ and Java Programming 3
Senior Membership 4
Southcon in Orlando a Success! 5
SoutheastCon 2005 – Teacher In Service Program Discussed 5
Sections Congress 2005 5
IEEE Meeting Calendar 6
Appling design patterns and object oriented methodologies to implement software rules 7

IEEE Canaveral Section
Newsletter

Vol. VI No. 1 http://www.ieee.org/canaveral January – April 2005

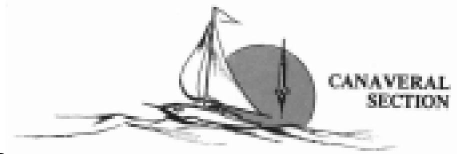
Adam (Buddy) Kissiah Gives Fascinating Lecture on Cochlear Implant Invention

The January 2005 meeting of the Canaveral Section featured a very interesting lecture by member Buddy Kissiah on the Cochlear Implant, which he invented while working for NASA and for which he received a patent. This invention not only helps restore hearing to people with certain kinds of hearing loss, but it has also been estimated by NASA to have a positive impact on the economy of billions of dollars. For this achievement, Buddy recently received a Space Act Award from NASA. Below is a photo of Buddy presenting his lecture at the Canaveral Section meeting.

Understandably, there was a lot of interest in attending this lecture by people with hearing impairment. For this reason, the Canaveral Section secured the services of a sign language interpreter, Sharon Goode, to provide real-time interpretation of Buddy's lecture. She did a remarkable job interpreting the sometimes complex and technical information presented by Buddy! One of the ironies of the evening was that due to some intermittent minor problems with the sound system, the hearing impaired attendees sometimes had a better understanding of what Buddy was saying than everyone else! But everyone agreed that Buddy and Sharon did a wonderful job with their presentation. A photo of Buddy greeting Sharon is shown below.

Joint Canaveral/Melbourne Life Member Chapter Holds First Meeting

The joint Life Member chapter of the Canaveral and Melbourne sections held its first meeting ever on March 16th at the Florida Institute of Technology in Melbourne. Buddy Kissiah gave an encore presentation of his lecture on the invention of the



IEEE Canaveral Section Newsletter

Vol. VI No. 1 http://www.ieee.org/canaveral January – April 2005

cochlear implant. Chapter chair Art Greene reported there were about 18 people in attendance, not bad for an inaugural meeting. If you are a life member of the IEEE and would be interested in helping organize future meetings, contact Art at angreene@juno.com.

NASA Engineer Ali Shaykhian Lectures on Design Patterns

Another interesting topic was featured at the March meeting of the Canaveral Section. NASA engineer Ali Shaykhian took time off from his prestigious teaching fellowship at Bethune-Cookman College to give a lecture on applying design patterns and object oriented methodologies to implement software rules on March 3rd at the Florida Solar Energy Center in Cocoa. Many people in attendance expressed interest in learning more about this topic so Ali has prepared a paper based on this presentation. The paper is included as an appendix at the end of this newsletter.

Canaveral Section To Offer Seminar on C++ and Java Programming

Vote for One or *Both* of these seminars!

NASA engineer Ali Shaykhian has offered to teach seminars on C++ and Java programming as a service to the Canaveral Section and the engineering community at large. Your **vote** will determine which of these two informative seminars (or both) will be offered so email me at ccmail@bellsouth.net or sec.canaveral@ieee.org and let me know which one (or both) of these seminars you'd like to attend!

The details of the seminars are under consideration but it is anticipated that they will be offered this spring or summer and will include credit for Continuing Education Units (CEUs). Members of the Canaveral Section will receive a discount on enrollment. The following paragraphs describe the topics that will be covered in these seminars:

C++ Professional Seminars

The C++ seminars are designed to examine the contemporary issues in C++. These seminars supplement the participants programming knowledge by focusing on object, object relationships, and polymorphism. Members may participate in all three seminars or pick and choose the topics of interest.

The C++ seminars teach the fundamentals of C++ programming language. The C++ fundamentals are grouped into two seminars that are offered in a consecutive order. Each Seminar is 4 hours long, which includes both lectures and practice discussions. The first seminar includes the functional aspect of C++ programming language, covers the essential framework of C++ programming language. The second seminar covers the object-oriented aspects of the language. This seminar provides the information necessary to evaluate various features of object-oriented programming, emphasizes on encapsulation, inheritance, and polymorphism.

Specifics goals and objectives includes:

- Create, compile and run C++ programs
- Read, recognize, and describe C++ syntax
- Write functions, decisions, loops and exceptions
- Differentiate among value, reference and pointer parameters
- Declare, define and use variables, constants, arrays, pointers and references
- Compare Data storage, stack storage, and heap storage
- Define and implement classes to represent real objects

Implement object-oriented designs, emphasize on encapsulation, inheritance and polymorphism.

Java Professional Seminars

The Java seminars are designed to teach fundamentals of Java programming language. No prior programming experience is required for participation in the seminars. Each seminar is 4 hours long for both lectures and hands-on practice.

The first seminar covers introductory concepts in Java programming including data types (integer, character, ..), operators, functions and constants, casts, input, output, control flow, scope, conditional statements, and arrays. The seminar also introduces introduction to Object-Oriented programming in Java, the vocabulary of OOP, relationships between classes, using existing classes, using packages, constructors, private data and methods, final instance fields, static fields and methods, and overloading.

The second seminar covers extending classes, inheritance hierarchies, polymorphism, dynamic binding, abstract classes, protected access. The seminar discussions additionally include interfaces, properties of interfaces, interfaces and abstract classes, interfaces and callbacks, basics of event handling, user interface components with swing, applet basics, converting applications to applets, the applet HTML tags and attributes, exceptions and debugging.

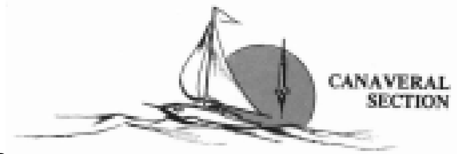
Please email me at ccmail@bellsouth.net or sec.canaveral@ieee.org if you are interested in attending one or both of these seminars.

Senior Membership

You too could qualify for Senior Membership in the IEEE (if I can do it, anyone can!). It looks great on your resume or your yearly accomplishments form at work. Senior membership requires ten years of professional experience, but your time attending college also counts (3 years for a bachelors degree, 4 years for a masters degree and 5 years for a Ph.D.). Promotion to Senior Member status also requires the recommendation of three existing Senior Members. This should also be a snap since quite a few members of the Canaveral Section have elevated their membership to Senior status in recent years and would be willing to serve as references. You can read more about the qualifications for Senior Membership at:

http://www.ieee.org/organizations/rab/md/smrequirements.html

If you're interested in pursing Senior Membership, write me at ccmail@bellsouth.net or sec.canaveral@ieee.org.



IEEE Canaveral Section Newsletter

Vol. VI No. 1 http://www.ieee.org/canaveral January – April 2005

SouthCon in Orlando a Success!

The SouthCon trade show, sponsored by the IEEE Florida Council was held in Orlando in February. I attended SouthCon for the first time this year and had enjoyable time seeing all the manufacturer's booths and attending the free seminars. The chairman of the show has reported that SouthCon reversed a trend of recent years by making more money than the year before, and 16 companies at this year's show have already signed up for next year. Let's hope this trend continues!

SoutheastCon 2005

Teacher In Service Program Discussed

I also represented the Canaveral Section SoutheastCon this month. Although I'd been to SoutheastCon before, this was definitely a learning experience for me in terms of opportunities for service to the section and the engineering community. In future newsletters I hope to write more about what I learned at SoutheastCon.

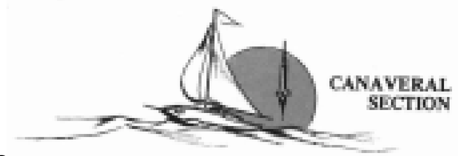
One opportunity I heard about in particular was the IEEE **Teacher In Service Program (TISP)**. The program features engineers developing and presenting technologically oriented topics to pre-college educators in an in-service or professional development setting. Teaching teachers about science and engineering could lead them to incorporate lesson plans on these subjects and encourage more students to pursue engineering as a career.

A workshop for IEEE members interested in participating in the TISP program is being held in Atlanta on July 23rd. Although the deadline for notifying the organizers of the workshop has passed, they have shown some leniency in continuing to accept the names of members who desire to participate since the workshop is still three months from now. If you have an interest in helping inspire students to be the engineers of tomorrow, send me email at ccmail@bellsouth.net or sec.canaveral@ieee.org.

Sections Congress 2005

The triennial IEEE Sections Congress will be held October 14th to 17th right here in Florida, specifically Tampa. This is a **big** deal because IEEE sections from all over the world will be sending representatives. The theme of Sections Congress 2005 is "Promoting a World Class Volunteer Community". Since it's being held so close by, I think the Canaveral Section should send as many representatives as possible. If you are interested in attending Sections Congress, send me an email at ccmail@bellsouth.net or sec.canaveral@ieee.org. For more information on Sections Congress, visit their web site at:

http://www.ieee.org/organizations/rab/sc/2005



IEEE Meeting Calendar

28 Apr 2005. "Patent, Trademark and Copyright and Trade Secrets: What Every Engineer and Tech Company Should Know About Intellectual Property"

Attorney Robert J. Sacco
Florida Solar Energy Center, Cocoa

Other activities are in the works. Please check the website regularly for latest developments.

2002 ExComm Officers:
Chuck Chapman - *Chair*
Ali Shaykhian - *Vice Chair*
Leon Migdalski - *Secretary*
Wayne Rendla - *Treasurer*
Chuck Chapman – AES/*Comp. Soc. Chapter Chair*
Dr. Bahman Motlagh – *Computer Society Vice Chair*

Some Important websites:
www.ieee.org
http://www.ewh.ieee.org/r3/canaveral/
http://www.ieee.org/membership/
Florida Council http://www.ieee.org/florida
SouthCon http://www.southcon.org
Sections Congress :
http://www.ieee.org/organizations/rab/sc/2005/

Appling design patterns and object oriented methodologies to implement software rules

Gholam Ali Shaykhian

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

Introduction

Functional decompositions were mainly used to design and implement software solution in the early era of computer programming. 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. Separation of function and data causes tight coupling between the two, meaning that a change in a data may require multiple changes to the design and code through the program. Tight coupling of data and function adversely may impact the cost of the software maintenance. Problems with Y2K can be noted as an example of tight coupling, the year 2000 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. Even though the fix was minimal; "just change the year from two digits to four digits", correcting the date error cost the business communities billions of dollar. The major cost was due to "tight coupling"; date was tightly coupled with all functions or procedures using the date; as such, changing the date required making several changes throughout the software program.

Object oriented methodologies

Object Oriented Methodologies (OOM) has evolved in the past two decades and now has dominated the software development process. Today, most software projects are planned and designed around the core artifact of object-oriented methodologies. This paper advocates the usage of object-oriented methodologies and design patterns as the centerpieces of software solution in implementing business rules. 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. An introductory programming example is introduced in C++ programming language to explore the usage of object-oriented programming and design patterns in connection with business rules. Key features of object-oriented methodologies are explained. 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

An object has attributes and behaviors, when design properly, remains as a cohesive entity; here we introduced it as object with cohesive characteristics. The necessary expectations of object with cohesive characteristics are: (1) object's attributes are not exposed to external entities, (2) object encapsulates only its own attributes and behaviors; unrelated data and functions are excluded from the object's definition, (3) changes to the state of an object are made through mutator member functions, and (4) access to attributes of an object are made through accessor member functions.

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 descriptions and characteristics of an object. 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 are 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 class Employee and class Customer are both inherit from (derive from) class Person and will have access to the protected section of the 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 Factory class. The constructor is placed in
// protected section of the class to disallow public access.
class Factory;          //forward declaration to allow dependency
class Person {
friend class Factory; // 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

Relationships among objects

Defining relationships among objects are an essential part of OOM. Relationships among objects are: is-a, has-a, knows-a, depends-a and other variations of these relationships (e.g. kinds-a, with-a, etc). Each of these relationships can be modeled to what we know of them in respect to a real life scenario or experience. For example, the is-a relationship can be modeled to design relationships between a child and a parent object, the has-a relationship describes an object being composed of or an aggregate of other objects, the knows-a relationship is used when a member function of an object requires interactions with other objects, and the depends-a relationship describes dependency among objects. The usage of these relationships are covered in the bank example, as follow:

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).**

The bank teller, manager and customer are objects of type Person; we read the model by saying "a bank teller is an employee", "an employee is a person", or "a bank customer is a person". All these form ***is-a*** **relationship.** ***Is-a*** **relationship forms strong relation among objects, known as inheritance. Inheritance among objects introduces two related equally important topics: (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 has access to the public members of class Person. Class Person is referred to as a base class, and objects of class Person are referred to as base objects, super objects, or parent objects. The class Customer, Teller or Employee is referred to as derived class, and the objects of derived class are referred to as derived objects, sub-objects, or child objects.**
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 non-static data and a derived object owns its class non-static data plus the non-static data of its corresponding base class.

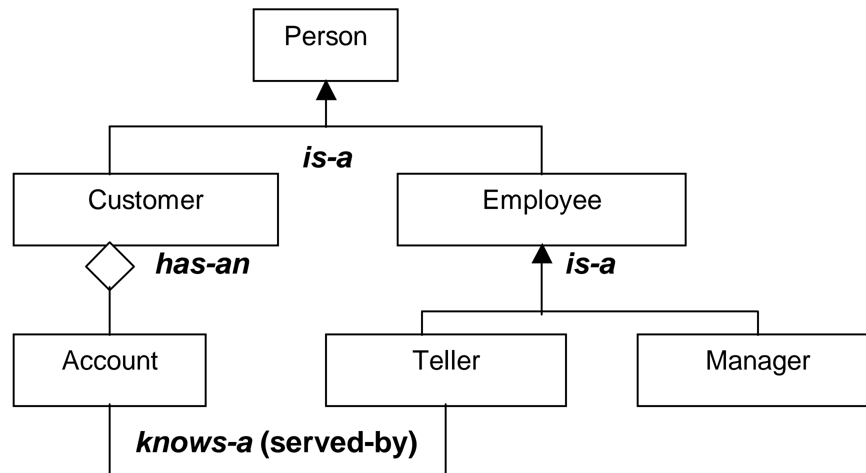


Figure-1 **Object relationships**

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

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. In this figure, dependency is established between Factory and Customer 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.

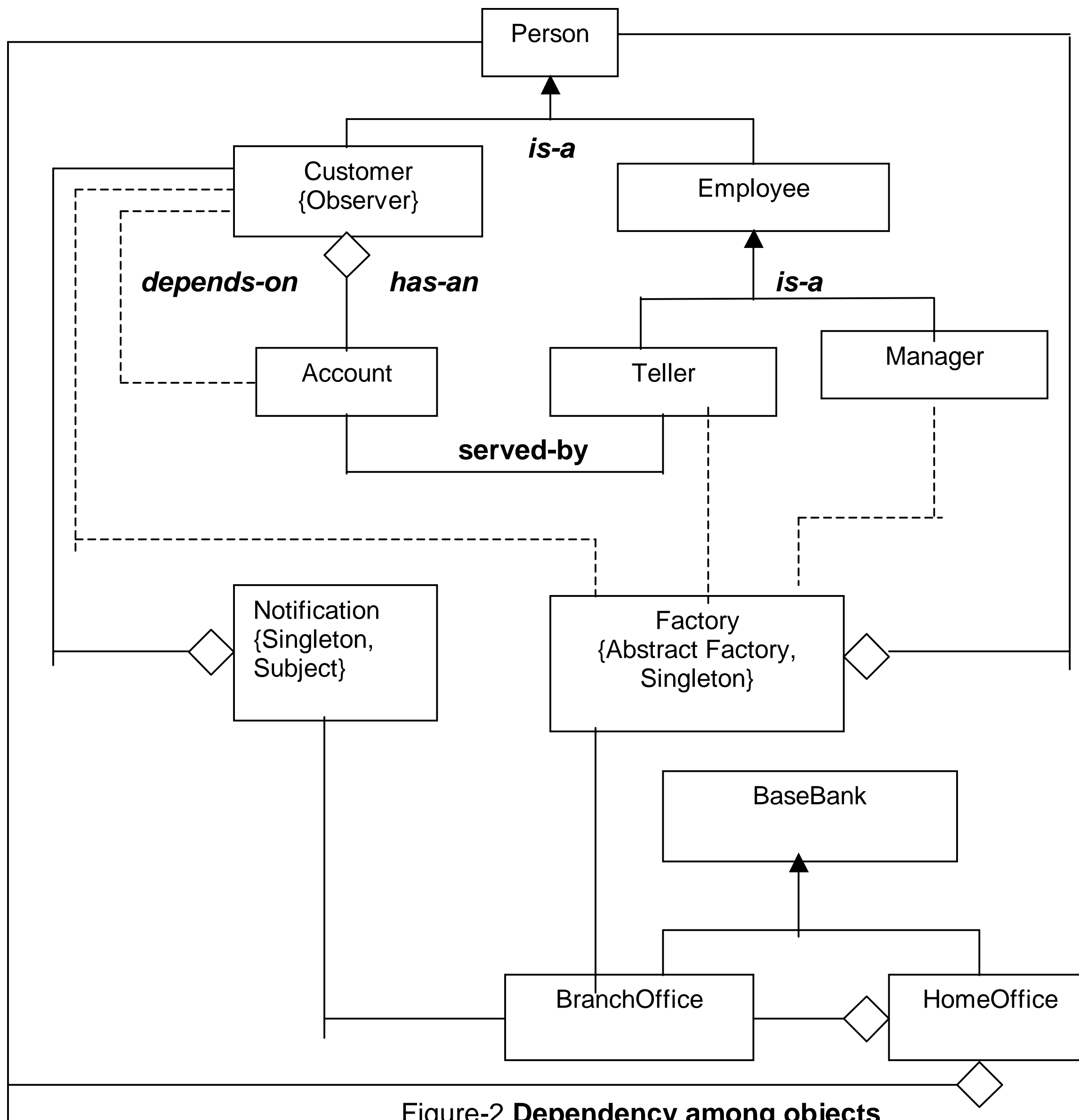


Figure-2 **Dependency among objects**

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 Customer (**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 Customer object then a call to show() would result Customer::show(). What is described here is known as polymorphism.**

```
// Customer.h <Header file>
#ifndef Customer_H  // multiple definition guard
#define Customer_H
#include "Person.h"  // for the base class
#include <map>        // for Customer's account
// Customer class is a derived class of Person class. The object creation of this
class
// is limited to Factory class. The constructor is placed in protected section of the
// class to disallow public access.
class Factory;            // forward declaration to allow dependency
class Account;                    // forward declaration to allow compilation
class Customer : public Person {
friend class Factory; // Dependency statement –to have access to the constructors.
public:
  virtual ~Customer();          // 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:
  Customer() {}
private:
  map<string,Account*> accounts;   // Customers, multiple accounts
};
#endif Customer_H

// Customer.cpp <implementation file>
#include "Customer.h"
#include "Account.h"
#include "Notification.h"
Account* Customer::createAccountObject(double d)
{
   return new Account(d); // create an Account() object
}
Account* Customer::getAccountObject(string &s)
{
  map<string,Account*>::iterator i;
```

```
  i = accounts.find(s);
  if (i!= accounts.end())
        return (*i).second;
  else
        return 0;
}
Person* Customer::getServedByObject(string &s)
{
  map<string,Account*>::iterator i;
  i = accounts.find(s);
  if (i!= accounts.end())
        return (*i).second->getServedBy();
  else
        return 0;
}
void Customer::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 Customer::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
}
Customer::~Customer()
{
  map<string,Account*>::iterator i;
  for (i=accounts.begin(); i != accounts.end(); i++) {
        delete (*i).second;     //avoid memory leak!
  }
  Notification *N=Notification::instance();
  N->unregisterCustomer(this);
}
```

Listing-2 **Description of class Customer**

Design patterns

Software development has always been an expensive undertaking. To this effect, the reuse of standard libraries and functions has always been a major part of the software development. “Design pattern” extends the reuse to reusing the design itself. Christopher Alexander says “Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice”. Even though Alexander was talking about patterns in

building and towns, what he says is true about object-oriented design patterns**.** The bank example uses Singleton, Abstract Factory, and Subject-Observer design patterns.

Singleton

The Singleton Design Pattern is used when exactly one instance of a class is required. The singleton object is created through the usage of static function data. The static function data once created is persistent throughout the program. The Singleton class provides a member function known as instance(), the instance() member function returns the singleton object to the caller. Every time the instance() member function returns the same (and the only) singleton object. **Listing-3** shows the code for the Singleton design pattern.

Abstract Factory

The Abstract Factory class provides a convenient way to centralize object creation of a suite of classes. **Listing-3** shows the implementation of both the singleton and abstract factory design patterns. The factory class assumes the responsibility to create a suite of related objects. The corresponding classes of these objects must limit their class object creation to factory class only. The code for working example describes the relationship of the family of related objects and the factory class.

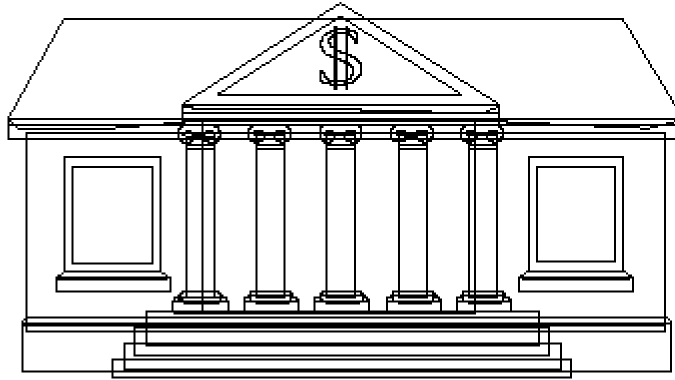
Subject-Observer

The Subject-Observer design pattern has two parts: subject and observer. The relationship between subject and observer is one-to-many. At any time, a subject can notify its observers. The working example uses the Notification class (subject) to notify its observes (the customers) when account balance falls bellow a predetermined amount. The working example shows the cod for subject-observer design pattern and its supporting member function to process the registration and de-registration of an account.

Working with the bank example

A C++ programming example provided to demonstrate the use of object-oriented and design pattern to implement business rule. 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 rules". Conducting a business rule through manual procedure may result in inconsistencies and could require significant resources. Alternatively, providing software solution to automate business rules may prove beneficial. Software solutions bear initial development cost, and thereafter maintenance cost.

The usage of object-oriented methodologies and design patterns as the centerpieces of software solution in implementing business rules is advocated throughout this paper. The program example incorporates a set of fictitious business rules listed in Figure 3 to advocate object-oriented solutions.



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.

Figure 3. **Examples of Bank Rules**

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 Customer 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 Customer 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.

```
// Factory.h <Header file>
#ifndef FACTORY_H
#define FACTORY_H
#include "Person.h"
// The Factory 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 Factory {
public:
  static Factory * instance();          // returns singleton object.
  Person * createCustomer();
  Person * createManager();
  Person * createTeller();
private:
        // additional attributes for Factory
};
#endif FACTORY_H

// Factory.cpp <implementation file>
#include "Factory.h"
#include "Customer.h"
#include "Teller.h"
#include "Manager.h"
Factory* Factory::instance() {
  static Factory theInstance;  // Singelton object
  return &theInstance;
}
Person * Factory::createCustomer() {  return new Customer(); }
Person * Factory::createManager() {  return new Manager();   }
Person * Factory::createTeller() {  return new Teller(); }
```

Listing-3 **Description of Factory 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 Customer; // Exclusive privilege to Customer 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 unregisterCustomer(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 "Customer.h"
std::list<Person*> Notification::customers;
Notification * Notification::instance()
{
  static Notification N;
  return &N;       // return singleton object
}
void Notification::notifyCustomer(Person *Customer,string accountType,
                       double amountAvailable, double amountWithdrawal)
{
   string message = "Non-Sufficent Funds <" + accountType+"> ";
   Customer *C = (Customer*)Customer;
   Person *accountServedBy = C->Customer::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::unregisterCustomer(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.
Customer provides a constructor with two string parameters (Listing-2). This constructor is used to create a customer object with name and identification.

The appendix section includes the related objects to examine the functionality of this program.

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 solve business rules are advocated and establishing cohesion within an object is emphasized. Tight coupling existed in functional decompositions was noted.

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

The Appendix section includes additional files that can be used to test the functionality of the Bank example.

```
// Employee.h <Header file>
#ifndef Employee_H // multiple defintion guard
#define Employee_H
#include "Person.h"  // base class
// Employee class encapsulates Employee data. The object creation of this class
// is limited to the Factory class. The constructor is placed in protected section of
the
// class to disallow public access.
class Factory;                    //forward declaration to allow dependency
class Employee : public Person {

friend class Factory;  // Dependency statement -to have access to constructor
```

```
public:
  ~Employee();          //  polymorphic  destructor
   virtual void show(); // object binds to this member function at runtime.
protected:
   Employee(const string &, const string &);         // disallow public access
private:
   // additional attributes for Employee
};
#endif

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

Listing-6 **Description of Employee class**

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

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

Listing-7 **Description of Teller class**

```
// Manager.h <Header file>
#ifndef Manager_H             // multiple definition guard
#define Manager_H
```

```
#include "Employee.h"
// Manager class inherits from the Employee class. The object creation of this
class
// is limited to Factory class.
class Factory;          //forward declaration to allow dependency
class Manager : public Employee {
friend class Factory; // Dependency statement - friend class
public:
  ~Manager(); // polymorphic destructor
  virtual void show();  // object binds to this member function at runtime.
protected:
  Manager(const string &, const string &);// construct object
private:
  // additional attributes for Manager
};
#endif

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

Listing-8 **Description of Manager 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 "Customer.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(Customer *);
//  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 "Factory.h"
#include "Notification.h"
#include "Customer.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(Customer *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((Customer*)(*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;
  Customer           *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 = (Customer*)(*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
//  Factory 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 "Factory.h"
#include "Person.h"
#include "Teller.h"
#include "Manager.h"
#include "Customer.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)
{
   Factory *BF =  Factory::instance();
   Person *P1= BF->Factory::createCustomer();
   P1->setName(S1);
   P1->setIdentification(S2);
   return P1;
}
Person* HomeOffice::addTeller(string &S1, string &S2)
{
   Factory *BF =  Factory::instance();
   Person *P1= BF->Factory::createTeller();
   P1->setName(S1);
   P1->setIdentification(S2);
   return P1;
}
Person* HomeOffice::addManager(string &S1, string &S2)
{
   Factory *BF =  Factory::instance();
   Person *P1= BF->Factory::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()
{
  // Test data 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();
  Teller *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<Teller*>(P);
                    if (T != NULL) B->BranchOffice::addTeller(P);
          }
          i++;
  }
}
void HomeOffice::assignManagers(BranchOffice *B) {
  string S = B->getBranchName();
  Manager *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<Manager*>(P);
                    if (M != NULL) B->BranchOffice::addManager(P);
          }
          i++;
  }
}
void HomeOffice::assignCustomers(BranchOffice *B)
{
  string S = B->getBranchName();
  Customer *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<Customer*>(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