



NEW IMPLEMENTATION OF EXAM-SCHEDULE MANAGEMENT SYSTEM (ESMS) IN OBJECT ORIENTED PROGRAMMING CONCEPTS USING C++ WITH UML DESIGNS

Brenda Lim Pei Pei

Student, BSc (Hons) Intelligent Systems, Asia Pacific University, Kuala Lumpur, Malaysia

ABSTRACT

This paper applied multiple object oriented concepts to develop the management system. The concepts employed here to increase the performance of management system by eliminating any code duplication and by increasing code reusability. Object Oriented principles have been achieved successfully hence abstraction, inheritance, polymorphism and encapsulation were enhanced based on its objects. Furthermore, other keywords, such as virtual and const are used so to increase the readability of codes and the maintainability of codes by keeping it clean and efficient. Error handling catches exceptions that occurred during runtime and that a composition is a strongest relationship type among association, aggregation and itself. Aggregation is considered to be weak because the child object has its own lifetime whereas composition stresses the lifetime of child object belongs to the parent and if the parent destructs, the child destructs as well. The advantage of this system explains about reusability and modularity so that the compilation time would be shorter and the management code would be much efficient when compared with structured programming approach.

Keywords: *Abstraction, Aggregation, Object Oriented, Inheritance, Polymorphism*

I INTRODUCTION

The object oriented concepts that have been implemented into the Exam-Schedule Management System are as listed below with sample screen captures of codes to further show how object oriented theories are applied into the system. Each code sample would be categorized under the classes where they belong for better references.

1.1 Abstraction—Header files and Source files

This section describes about how classes are used to categorized different code segments into a separate structure for better management and code monitoring. Each classes created contained data members that are private and public and in some instances, protected members as well—Staff class and Validation class. Multiple program-defined libraries are included into the listed classes for better utilization of pre-define functions, such as fstream, tuple and



iostream. Programmer defined libraries are differentiated with a “” instead of a <>, such as #include “Professional.h” and #include <tuple>.

All classes created are stored in a header file and a source file. Header files are used to declare data members and member functions which are later defined in the source files. Header guards are utilized so to prevent redefinition error that might occur during program compilation. Whenever a source file compiles, the program will go through all the included files so it is possible for a header file to be appended into build several times. In simple terms, in order to counter redefinition error, we say if the header file is yet to be included, define it, else do not define. Such compilation error usually happens when classes share data members or functions with each other through inheritance and friend function.

Abstraction is applied to seven classes, which are Certificate, Exam, ExamSchedule, Professional, Specialism, Staff and Validation. All seven classes are separated into header class, for function declaration and source files, for function definition. Sample shown below are the Staff class, Validation class and on

Certificate

```
#include <tuple>

#include "Professional.h"
#include "Specialism.h"

#ifdef Certificate_H
#define Certificate_H

using namespace std;

class Certificate { ... };

#endif
```

1.2 Encapsulation

Classes that utilizes accessor functions are Certificate class, Exam class, Professional class, Specialism class and Staff class. All five classes uses constant for their getters and that a variety of data types are used to overload the setters' parameters. The data types included in parameter overloading are string, integer, character and double. A return type is required for all getters so to return the retrieved value to the requesting user. Sample code shown below are the Professional class and Specialism class.

Professional



```
void setPID(string, int, int);
void setPIDExam(string);      string getPID() const;
void setGender(char);         char getGender() const;
void setNric(string);         string getNric() const;
void setContact(string);      string getContact() const;
void setAddress(string);      string getEmail() const;
void setEmail(string);        string getAddress() const;
```

Specialism

```
void setSpec(string);
string getSpec() const;
```

1.3 Inheritance

Inheritance between Validation and Staff with Professional is multiple inheritance and that the sharing of information between Professional class and Certificate class is of single inheritance. Therefore, the program utilizes a hybrid inheritance structure which composes of both multiple and single inheritance from the four listed classes. Staff class shares data member, name with Professional and the same happens to Professional with Certificate. The function members required to set and get names are shared across the three as well. Moreover, Validation class shares its protected function members with Professional class so that whenever user creates a new professional, data validation could happen via accessing the shared methods. Sample shown below are the four mentioned classes, Staff, Validation, Professional and Certificate.

Staff—Parent to Professional

```
#include <iostream>
#include <cstdlib> //system("pause")
#include <string>

#ifndef Staff_H
#define Staff_H

using namespace std;

class Staff
{
protected:
    string name;

private:
    string username;
    string password;

public:
    Staff();
    ~Staff();

    void setName(string);
    void setUsername(string);
    void setPassword(string);

    string getName() const;
    string getUsername() const;
    string getPassword() const;
};

#endif
```



Validation—Parent to Professional

```
#include <iostream>
#include <regex>

#ifndef Validation_H
#define Validation_H

using namespace std;

class Validation
{
protected:
    bool formatName(string *x);
    bool formatNric(string *x);
    bool formatNumber(string *x);
    bool formatEmail(string *x);
    bool formatGender(char *x);
};

#endif
```

Professional—Child to Staff and Validation, Parent to Certificate

```
#include <fstream>
#include <tuple>
#include <limits> //numeric_limits for input validate
#include <cstdio> //rename for text file //fflush
#include <cstdlib> //system("pause")
#include "Staff.h"
#include "Validation.h"

#ifndef Professional_H
#define Professional_H

using namespace std;

class Professional :public Staff, public Validation
{
    _
```

The inclusion of parent classes are as shown along with the declaration of Staff and Validation as parents with an access modifier of public so that all data shared is a copy of their original access modifier though private members remain invisible and inaccessible by child.



Certificate—Child to Professional

```
#include "Professional.h"
#include "Specialism.h"

#ifndef Certificate_H
#define Certificate_H

using namespace std;

class Certificate : public Professional
```

The inclusion of parent class are as shown along with the declaration of Professional as parent with an access modifier of public so that all data shared is a copy of their original access modifier though private members remain invisible and inaccessible by child [1].

1.4 Polymorphism

This section describes the use of runtime polymorphism and compile time polymorphism in the program. Runtime polymorphism, also known as dynamic polymorphism is the overriding of member functions whereas compile time polymorphism, also known as static polymorphism is the overloading of member functions [2]. Both cases are employed in classes, such as Professional and Certificate.

1.4.1 Runtime Polymorphism—Override

Overriding can happen when a child calls a function declared and defined in the parent class as shown in the figures below. Although the parent object is assigned to the child's, during runtime, the program would be capable of deciding which class to refer to, in this case, the child's setName() would be called.

Professional

```
Staff *sp = new Professional();
sp->setName(name);
```

Certificate

```
Professional *pc = new Certificate(spec);
pc->setName(name);
```

1.4.2 Compile time Polymorphism—Overload

Overloading can happen to constructor and member functions by manipulating their argument list. It is mainly used to increase the reusability of codes and to eliminate code duplications from happening. The three most common



overloading schemes are the sequence of data types, the data type used and the number of parameters in the argument list [3]. Sample shown below partly covers the instances of static polymorphism written in the program.

1.5 Constructor

Sample shows the usage of member initializer syntax which could increase the efficiency of code compilation by reducing the compilation time. The usage of composition can also be seen at the overloading of constructor, in this case, of type Specialism. More detail about how composition is used within the program would be discussed later.

```
[-] Certificate::Certificate()  
    :CID(""), uniqueCID(0), description(""), fee(0.0)  
{  
    name = "";  
}  
  
[-] Certificate::Certificate(Specialism a)  
    : spec(a), CID(""), uniqueCID(0), description(""), fee(0.0) //member  
      initializer syntax (MIS)  
{  
    name = "";  
}
```

1.6 Composition

Composition is a stronger relationship type between classes when compared to aggregation and association. Child objects do not have their own lifetime and are dependent to the creation and deletion of the parent object [4]. For an instance, the creation of Specialism depends on Certificate. Specialism is an element that could only be invoked by creating a Certificate. Without it, Specialism would not stand any definitive meaning hence Specialism is tied to the lifetime of Certificate and could only survive as long as the parent lives. The parent has to include the child's header file and to declare the child's object as a member in the header file. As for source file, the accessing of data members and member functions from child could be done with a dot (.) because child objects created are stored in stack and could only live within scopes. Sample shown below are part of how composition is being implemented into the system.

1.7 Aggregation

Aggregation is a relatively weaker relationship between parent and child when compared to composition but the child has its own lifetime and does not survive by depending on the parent's lifetime. The parent could claim ownership over the child object and that once the parent has been destroyed, the child could still live on until the user calls for a child object destruction. As an example, ExamSchedule being the parent of Professional. When ExamSchedule is created, it could freely access data members and member functions of Professional, sans private data members, because it is capable of claiming ownership over what is owned by the child class though the child



would be able to live on even when the parent dies, further contrasting the freedom of lifetime between a child of aggregation and a child of composition. The parent has to include the child's header file and to declare the child's object as a member in the header file. As for source file, the accessing of data members and member functions from child could be done with an arrow (->) because child objects created live in the heap, which could only die when its class destructor is called. Sample shown below are part of how aggregation is being implemented into the system.

II KEYWORD

2.1 Virtual

This keyword could only be used whenever there is a child class calling for a modification to parent's member function. For an example, Professional is the parent to Certificate and that both the classes share a function called booldupCheck(string *) which returns a boolean and takes in a pointer of type string as a parameter. The function would be executing differently depending on the nature of the class data members and in this case, Professional and Certificate would have a totally different execution approach to the function. Professional would use the function to check for professional duplications in text file whereas Certificate would use it to check for certificate duplications in text file. In conclusion, virtual is used to inform the compiler that there would be modifications made to the function in the child's class and could be understood as being similar to @Override in Java.

2.2 Error Handling

This section discusses how errors are handled in the program, typically errors associating to the opening of text file and the validation of user input. Try and catch are used across three different classes and the main source file to handle exceptions thrown that are related to the streaming of text files (ifstream), such as the reading and opening of text file and the regex comparison of user inputs, such as the validating of contact number. An error message will be thrown once an exception on the opening or streaming of text file has been thrown or once an exception on the input format validation has been thrown so to inform the user about the issue. Sample shown below are two of the few source files implementing error handling.

3.0 Design Solution

This section would be explaining on the design strategy behind the creation of each classes, including the main source file. The description of each classes will be broken down into 3 different parts, the header file, source file and text file for better understanding. Each explanation is made with a sample visual representation that partly represents the whole class.

Main

Main is added in with a few global variables and a few lines of function prototypes which would be later defined in the bottom part of the source file. The setting of console title is defined with the inclusion of the <windows.h>

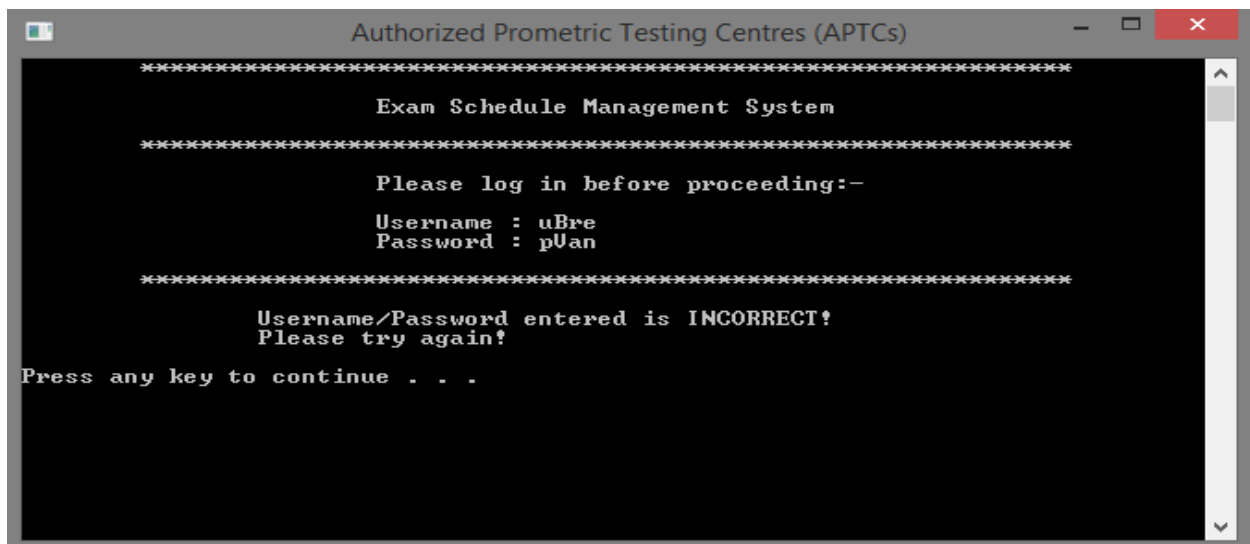


library and with the same library, the employment of #define NOMINMAX is made so to inform the compiler that the pre-defined max() function is not required within the validation of integer input. The program-defined max() comes with a set of parameter in the argument list but the integer input validation requires only the compiler to check the maximum stream size of user input but not to compare values between integers hence the definition of NOMINMAX has to be made before entering the main scope.

3.1 Screenshots of output with Explanations

Name of page: User login

Description: User are to enter a valid set of username and password in order to proceed with the system, else an error message would be prompted.



```
Authorized Prometric Testing Centres (APTCs)

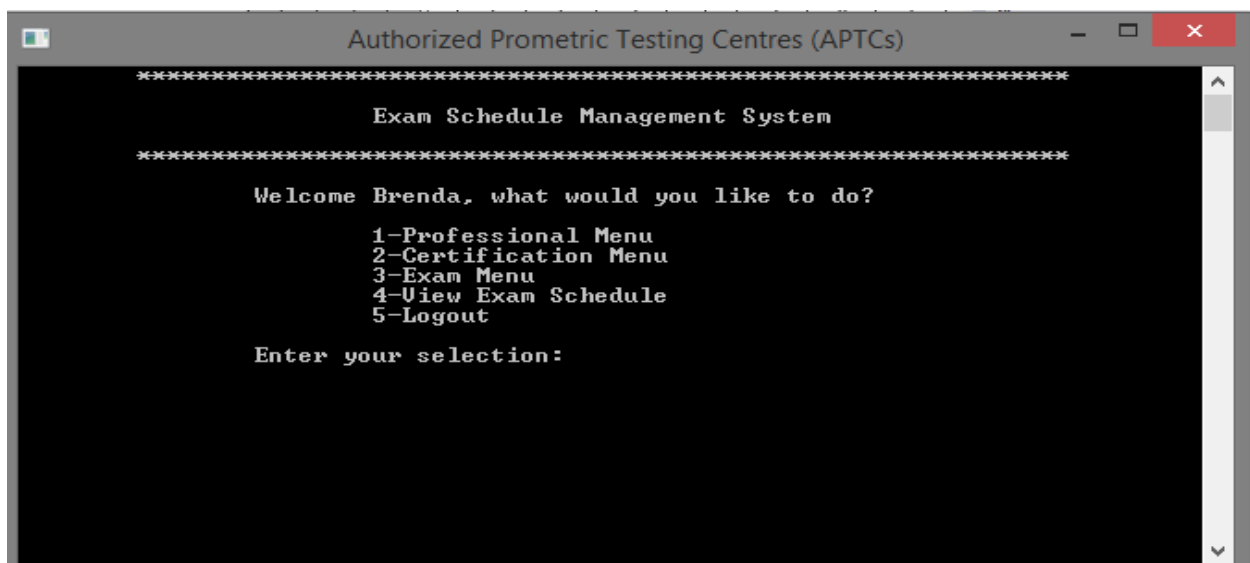
*****
Exam Schedule Management System
*****

Please log in before proceeding:-
Username : uBre
Password : pUan

*****

Username/Password entered is INCORRECT!
Please try again!

Press any key to continue . . .
```



```
Authorized Prometric Testing Centres (APTCs)

*****
Exam Schedule Management System
*****

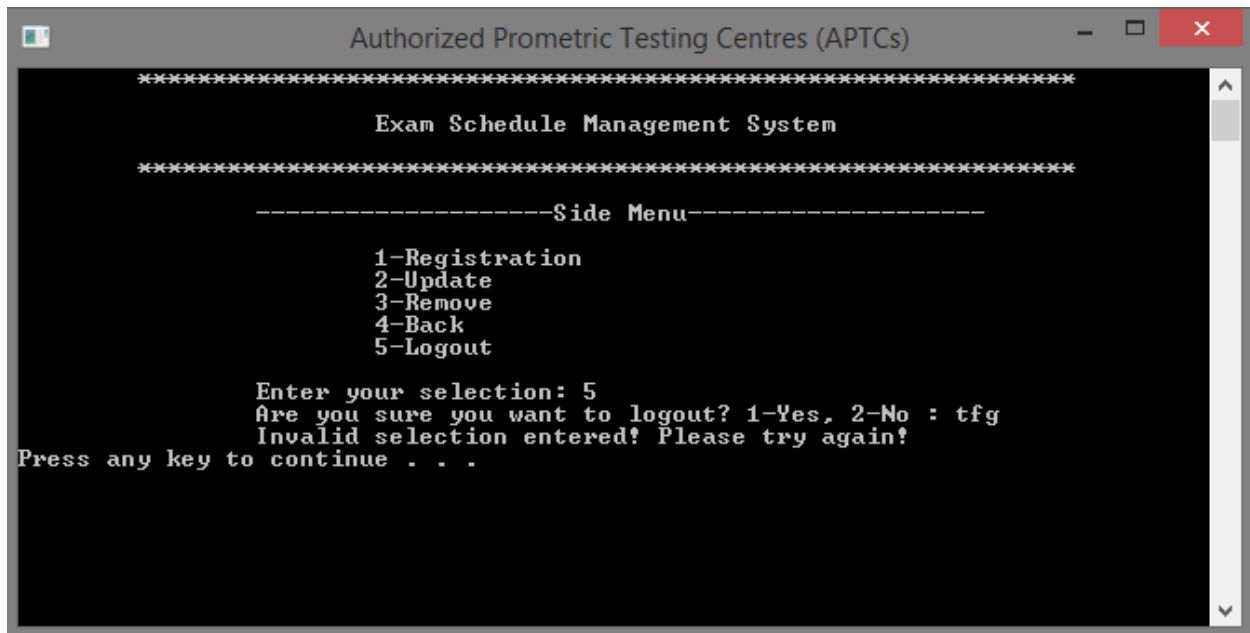
Welcome Brenda, what would you like to do?

1-Professional Menu
2-Certification Menu
3-Exam Menu
4-View Exam Schedule
5-Logout

Enter your selection:
```


Name of page: Main Menu

Description: User are to proceed with any of the suggested operations. Illegal input would be prompted an error message. Users would be redirected to this page upon the succession of performing any of the listed options.



Name of page: Side Menu-Logout

Description: User would be prompted a confirmation message upon selecting 5 of whether to really logout from the program or to stay logged in. 1 indicates to logout from the program whereas 2 indicates to stay logged in. Selecting 1 would the user be redirected back to the login page. Selecting 2 would the user be redirected back to the side menu. Invalid input would the user be prompted error message.

IV UML DIAGRAMS

For the implementation of Exam-schedule management system different UML diagrams were used under system model. But this article covered with part of the class diagram. The following “Fig.1” shown part of the classes such as validation, staff, professional and exam. These classes includes both attributes and functions. Several relationships were identified between them. Some of the classes interrelated with association relationship, some shown inheritance relationship and so on. In the usecase diagram this implemented system followed the proposed usecases as functionalities. Activity diagrams also followed to describe the functionalities of implemented system.

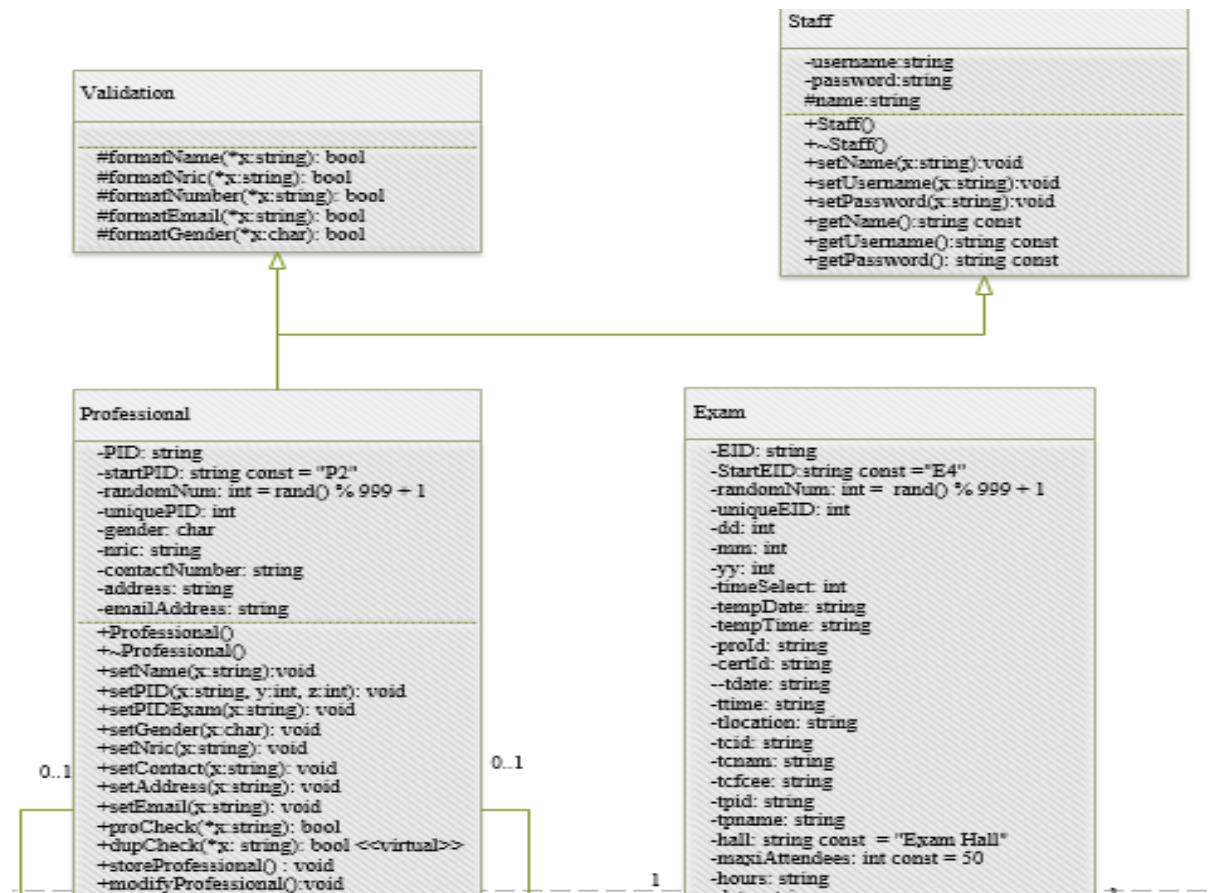


Figure 1: Class Diagram

4.1 Additional Features

a. Staff

- Users would be able to log into and log out from the system.

b. Professional

- Users would be able to modify and remove existing members.

c. Certificate

- Users would be able to modify and remove existing certificates.

d. Exam

- Users would be able to modify and remove existing exam slots.



V LIMITATION

The five limitations that I have compiled for this Exam-Schedule Management System are as listed below:-

a. Allow professionals of other nationality to register for exams

System is recommended to enable options for professionals of other nationality to register for exams.

b. Allow a third gender option for professional registrations

System should accommodate options for a third gender so to respect gender equality.

c. Expand the number of sponsors available

System should include options of adding in sponsors so to attract more professionals to partake the listed exams.

d. Include more exam schedule viewing options

System is suggested to include more exam schedule viewing options, such as via certificate ID or via a particular user input date.

e. Include staff account management

System is recommended to include a function that could sort out staff accounts via interface without having the technician to perform backend updates.

VI CONCLUSION

I have applied multiple object oriented concepts into the management system, ranging from the popular four, abstraction, inheritance, polymorphism and encapsulation to error handling. These concepts are employed so to enhance the performance of the system by eliminating any code duplication and by increasing code reusability. Inheritance is the sharing of data members and member functions from the parent class to the child class though private members remain invisible and inaccessible by child classes. Friend could be used to break this problem and is a recommended use whenever there is a need to share only one or two functions across several classes. These listed object oriented approaches focuses heavily on improvising data maintenance and data manipulation, such as bug fixes and code upgrading. Furthermore, other keywords, such as virtual and const are used so to increase the readability of codes and the maintainability of codes by keeping it clean and efficient. Error handling catches exceptions that occurred during runtime and that a composition is a strongest relationship type among association, aggregation and itself. Aggregation is considered to be weak because the child object has its own lifetime whereas composition stresses the lifetime of child object belongs to the parent and if the parent destructs, the child destructs as well. In conclusion, the application of object oriented concepts in the management system has high-lighted the importance of code reusability and modularity so that the compilation time would be shorter and that the management of code would be much efficient when compared to structured programming approach.



VII ACKNOWLEDGMENT

The author would like to share gratitude to Mr Umapathy Eaganathan, Lecturer in Computing, Asia Pacific University, Malaysia also Miss Angel Aron for her constant support and motivation helped me to participate in this International Conference and also for journal publication.

REFERENCES

- [1] Hidayat, A., 2015. *C++ Multiple Return Values*. [Online] Available at: <http://ariya.ofilabs.com/2015/04/c-multiple-return-values.html> [Accessed 2 4 2016].
- [2] Singh, C., 2014. *Method Overloading in Java with examples*. [Online] Available at: <http://beginnersbook.com/2013/05/method-overloading/> [Accessed 29 4 2016].
- [3] Singh, C., 2014. *Types of polymorphism in java- Runtime and Compile time polymorphism*. [Online] Available at: <http://beginnersbook.com/2013/04/runtime-compile-time-polymorphism/> [Accessed 29 4 2016].
- [4] Varun Gupta, Ajay, 2014. *Difference between association, aggregation and composition*. [Online] Available at: <http://stackoverflow.com/questions/885937/difference-between-association-aggregation-and-composition> [Accessed 29 4 2016].