FUDMA Journal of Sciences (FJS)
ISSN online: 2616-1370
ISSN print: 2645 - 2944
Vol. 7 No. 2, April, 2023, pp 57 - 64
DOI: https://doi.org/10.33003/fjs-2023-0702-1710

COMPARATIVE ANALYSIS OF AGGREGATION AND INHERITANCE STRATEGIES IN INCREMENTAL PROGRAM DEVELOPMENT

[*1]Godwin A. Otu, [2]Suleiman A. Usman, [3]Raphael U. Ugbe, [1]Stephen E. Iheagwara ,[1]Akudo C. Okafor, [4]Fredrick I. Okonkwo, [1]Oyebanji M. Shukurah, [5]Adeniyi U. Adedayo

[1]Department of Computer Science, Air Force Institute of Technology Kaduna, Nigeria.
[2] Department of Cyber Security, Air Force Institute of Technology Kaduna, Nigeria
[3]Department of Physics, Nigerian Defence Academy Kaduna, Nigeria.
[4]Department of Information and Communication Technology, Air Force Institute of Technology Kaduna, Nigeria.
[5]Information and Communication Technology Centre, Air Force Institute of Technology Kaduna, Nigeria.

*Corresponding authors' email: goddy2fine@gmail.com

ABSTRACT

Programming computers has been a herculean task for most programmers especially when codes grow into complex and larger software systems with multiple subprograms. Object Oriented Programming (OOP) has reduced the difficulty in the development of elegant and scalable software by presenting robust concepts such as composition, inheritance and aggregation. All these concepts have enormous assistance to the software developer in code reuse. Also these techniques can be used to build applications which can be delivered to customers in a record time. In this research a critical study, review and implementation of software building and enhancement using aggregation and inheritance. A module is built with attributes defining it properties and methods its characteristics. Incrementally more modules were added to the previous modules using either aggregation or inheritance technique. This incremental approach has proven tremendous success in software development. This as buttressed by many software development theories have shown that: software is built not manufactured; software is a collection of programs with functions and attributes based on its enhancement, also incremental software development which involves building systems from sub-systems gives a better understanding of software development process. Also the process of writing bug-free programs can be achieved with lesser difficulty which can be achieved when programs are built using modular or incremental software development approach, which employs mostly aggregation while moderately using inheritance only if all the properties and methods of those modules are needed wholesomely in classes. The result from the research will help programmers to enhance codes with much mastery.

Keywords: Aggregation, Composition, Incrementally, Inheritance, Module, Object

INTRODUCTION

The design and building of computer programs, subprograms and elegant software has laid down techniques drawn from software engineering and the choice of programming language paradigm used for implementation. This approaches when followed and implemented in sequence of steps based on the designated algorithm and the construct of the programming language will produced easy to use and well documented software. Being able to identify the tools and techniques to use and when to apply will assist the software engineer or programmer to define project delivery time, able to deliver software product and also the needed tools for software development projects. Object oriented programming (OOP) consists of a set of objects, which can change dynamically, and which can execute by acting and interacting with each other, in much identical way as a real-world system operates strictly guided by it properties and functionalities. It makes programs more intuitive to design, faster to develop, less difficulty during modifications processes, enhancing code reuse to save programming time and most importantly easier to understand codes. In the object-oriented view of programming, instead of programs consisting of sets of data loosely coupled to many different procedures, programs consist of software modules called objects that encapsulate both data and methods while hiding their inner complexities from software developers (Asagba and Ogheneovo, 2010). The concept of incremental programming, inheritance and aggregation and composition is worth describing and knowing because mastering these concepts strengthen the proficiency of a programmer. A module is a subprogram that is used to solve a specific task in a software. Many programs can be decomposed into a series of identifiable subtasks. It is a good programming practice to implement each of these subtasks as a separate program module. Hutabarat et al. (2009) described the idea of modular programming is to sub-divide a program into smaller units that are independently testable and that can be integrated to accomplish the overall programming objective. The use of modular programming structure enhances the accuracy and clarity of a program, and it facilitates future program alterations. Inheritance is the ability of an existing class to create new classes. Thus existing class is referred to as a base class and the newly created classes are called derived class. The derived class inherits all the properties inherent in the base class. Afolorunso and Vincent (2020) opined that inheritance is perhaps one of the most powerful features of object-oriented programming paradigm and also buttressed that inheritance can support program (or software) reuse, reliability, and modification of the base class. Inheritance is a powerful programming tool and it supports reusable component. Inheritance establishes a parent-child relationship (Asagba and Ogheneovo, 2010). Composition on the other hand is a relationship between two classes that is based on the aggregation relationship. Composition takes the relationship one step further by ensuring that the containing object is responsible for the lifetime of the object it holds. If Object B is contained within Object A, then Object A is responsible for the creation and destruction of Object B. Unlike aggregation, Object B cannot exist without Object A (Afolorunso and Vincent, 2020).

Modularity is a key concern in programming. However, programming languages remain limited in terms of modularity and extensibility. Small canonical problems, such as the Expression Problem (EP), identify some of the basic issues: the confusion between choosing one extensibility over another one in most programming languages. Other problems, such as how to express dependencies in a modular way, add up to the fundamental issues and remain a significant challenge (Weixin et al., 2021).

In this research program modules has been built using classes which will comprise instance variables and methods which will change the state of those variables. Then the concept of aggregation of objects will be used to incrementally build larger programs from already built classes, also generalization will be used to identify when to use inheritance between classes.

The act of building, coupling and testing program modules has been simplified with the introduction of object-oriented programming approach. Applying concepts like encapsulation, data abstraction, data hiding, inheritance and aggregation or composition have remove a lot of difficulty software building, testing, debugging, maintenance, documentation and most important of all understanding of software and usage by users or customers (Prehofer, 2001). Many researchers have work on various methods of code building using object-oriented approach.

A comprehensive introduction to inheritance, starting from its history and conceptual background, studying its target and actual usage, and also analyzing its importance in light of the current knowledge. More so the different types of inheritance were analyzed, and a simple taxonomy of inheritance mechanisms was presented in the study (Taivalsaari, 1996). Klump (2001) provided a concise preliminary to the concepts and advantages of the objectoriented approach and describes why power engineering students may benefit very much from a more formal introduction to OOP. Instead of a rigid class design, a method which involves writing features which are composed appropriately when creating objects was proposed. The new model for flexible composition of objects from a set of features which are services of an object and are similar to classes in object-oriented languages. In many instances, features have to be adapted in the presence of other features, which is also called the feature interaction problem. Noble and potter (1997) described a program monitoring method which takes account of aggregation and aliasing, and which can be used to detect changes automatically. Automatic change detection can simplify programming and design, so producing more reliable systems with less energy. Zeynab (2015) researched and published a comprehensive literature review over relationships among objects and also identified three basic types of relationships, including generalization or specialization, aggregation and association. (Weixin et al., 2021) presented a new statically typed modular programming approach called Compositional Programming. It is very easy to get extensibility in different dimensions. Compositional Programming gives an alternative way to model data structures that differs from both algebraic data types in functional programming and conventional OOP class hierarchies as modules of systems and a programming environment designed to support interactive program development in Scheme. The module system extends lexical scoping while maintaining its flavor and benefits and supports mutually recursive modules. The programming environment supports dynamic linking, separate compilation, production code compilation, and a window-based user interface with multiple read-print contexts (Hutabarat et al., 2009). Tung, (1992) proposed a theoretical foundation of module and modular programming: formal, proven, and easy to understand definition of module; modular programming, and module-based encapsulation. The theory is tested using source-code of many programming-languages. A concept module-based encapsulation and programming language named Nusa that eases the comprehension of module, program, modular-programming, and module-based encapsulation was developed. Otu et al. (2022) showed that an algorithm can be modified and applied to solve varied problems even in different problem domains.

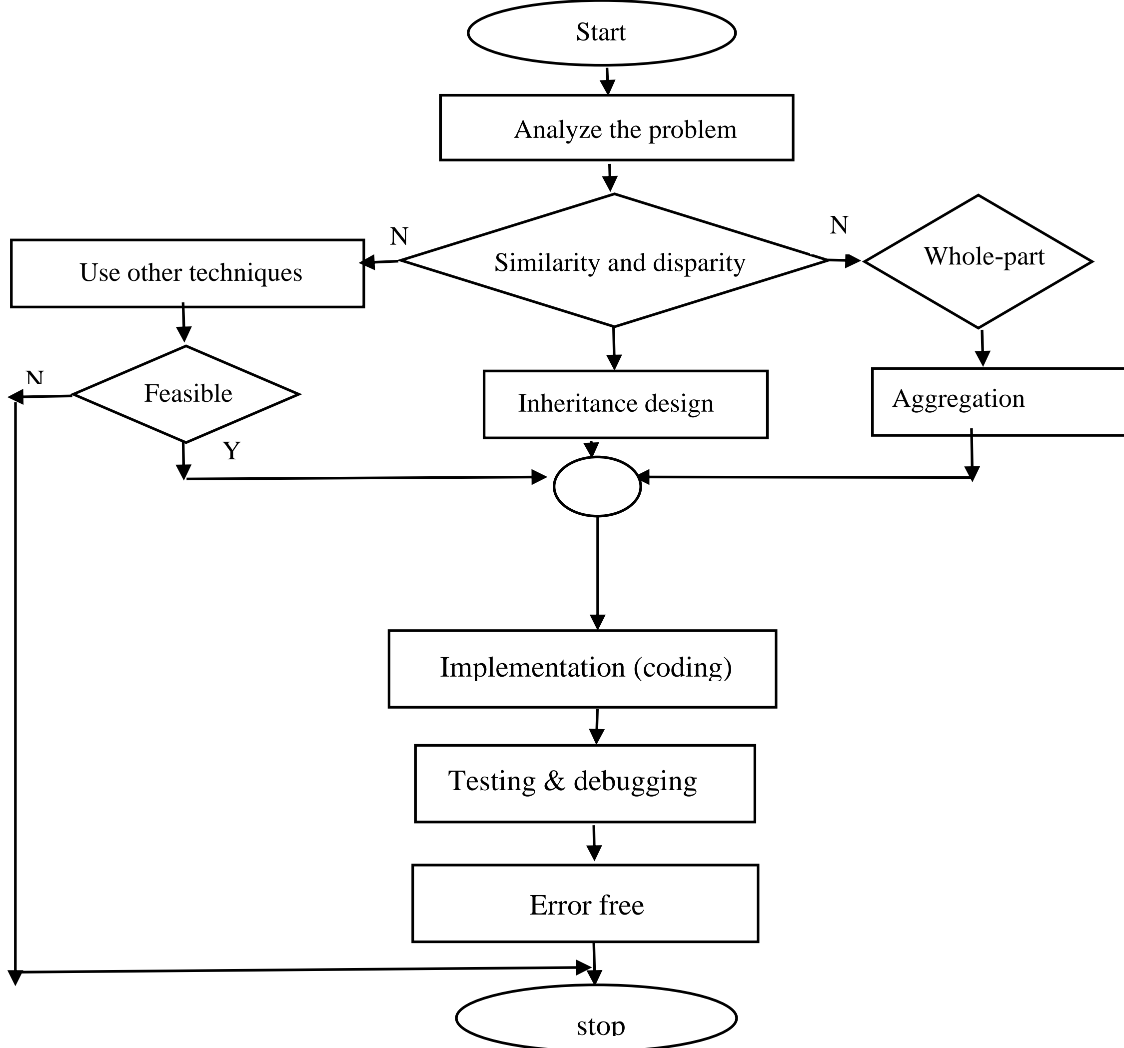
The concepts of aggregation, composition, generalization and specialization will be harnessed in this research to enable programmers to understand when to use these concepts, the advantages and demerits and why some of these concepts should be used with caution and moderation, and also why some should be encourage and others discouraged except in the absence of an alternative. We shall see how aggregation/composition can easily be used to incrementally add to code in terms of enhancing the code and increasing complexities by adding the classes or modules with less difficulty. The research addresses when to either use aggregation or inheritance in software development.

METHODOLOGY

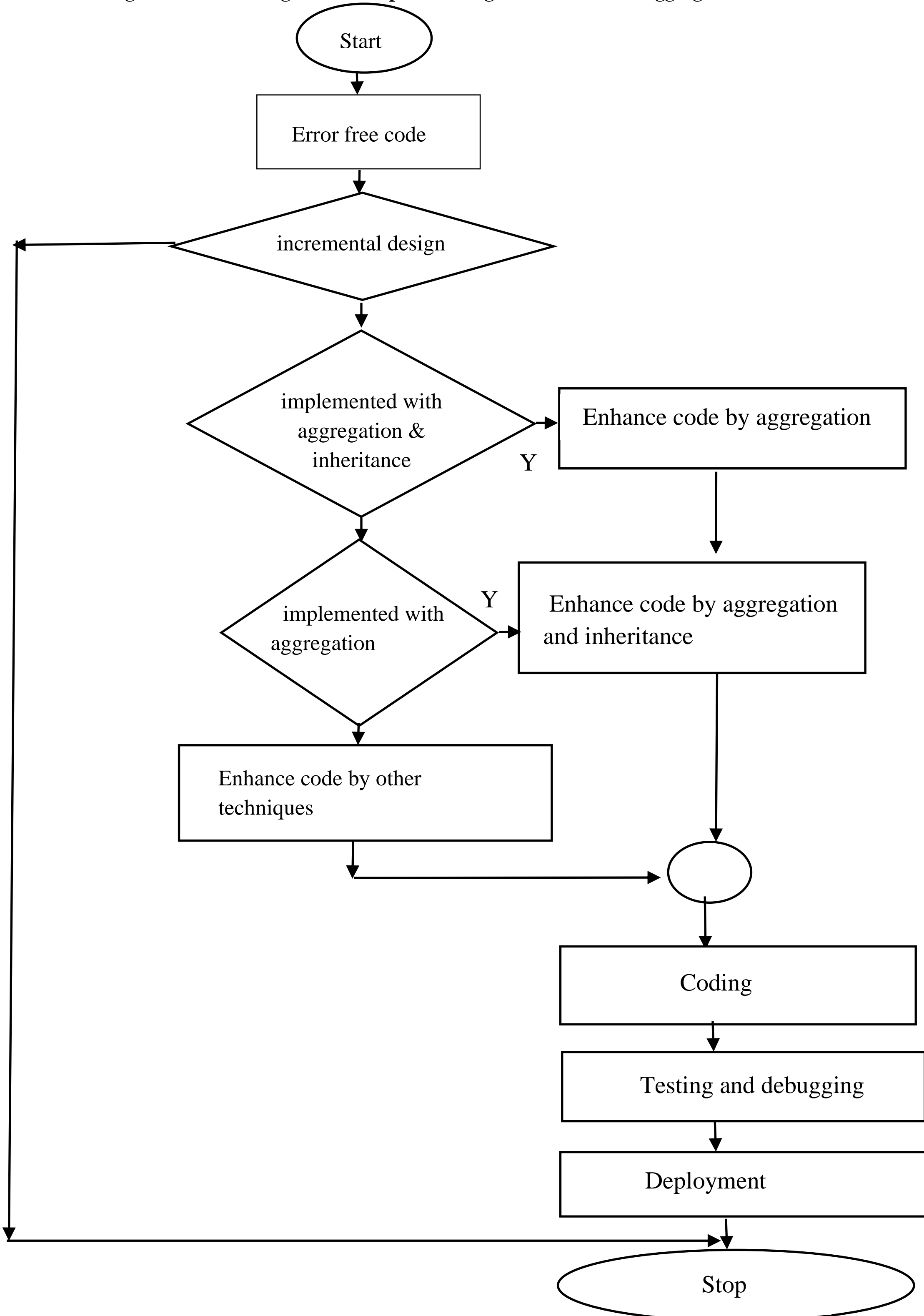
The research adopts similarity, disparity and whole-part techniques for the identification, formulation and modeling of objects. The objects are modeled using UML (Universal Modeling Language) scheme, which clearly describes both the state and behavior of the objects. The next step is the implementation (coding) of the design, testing and deployment. Given a programming task to solve using object-oriented programming approach the objects states and behaviors are studied, if the objects have similarities and disparities in terms of their instance variables (properties) and functionalities, then inheritance is applied to group the objects. The classes formed with similarities to the other classes are called generic, super or parent classes, while the classes with disparities that also have the properties and behaviors of the parent classes are called either derived or subclasses. The similarity and disparity technique is very useful in identifying inheritance among classes. The whole-part approach defines a class or object consisting of other aggregate of objects loosely coupled in aggregation and tightly coupled in composition. So, when an object can contain other objects used as its instance variables then we apply aggregation a loose form of composition.

The research also states the reasons why aggregation should be mostly applied in software development, and also states why inheritance should be moderately used in any software development process. Subprograms will be built, debugged, tested and the aggregated to a class to form a whole. This incremental approach will reduce complexity as the software grows and makes bugs identification and removal easy. Also shown in the complexity that grows with multiple inheritance. The methodology clearly analyzes how to group objects for inheritance in terms of specialization and generalization, and also how to enhance programs into complex software using a less complicated and easy to use method called aggregation. Also describes elaborately the significance of aggregation over inheritance. Also present scenarios when inheritance becomes unavoidable. The methodology diagrammatically depicts the various steps using well labeled flowcharts to analyze the processes of software processes.

Flowchart to Schematically Depict the Classification of Objects



Flowchart Showing Incremental Program Development using Inheritance and Aggregation



APPLICATION OF THE METHODOLOGY

A person object has first name, second name and third name, a civil servant is a person with a job designation and a salary and consist of a car object which has model, type and name, and also an address object which has a house number, street name and state of residence. A student is a person but with a faculty, department and matric number. If both objects consists of a birthdate object having year, month and day. The scenario can be modeled using the similarity, disparity and whole-part methodology used in this research thus:

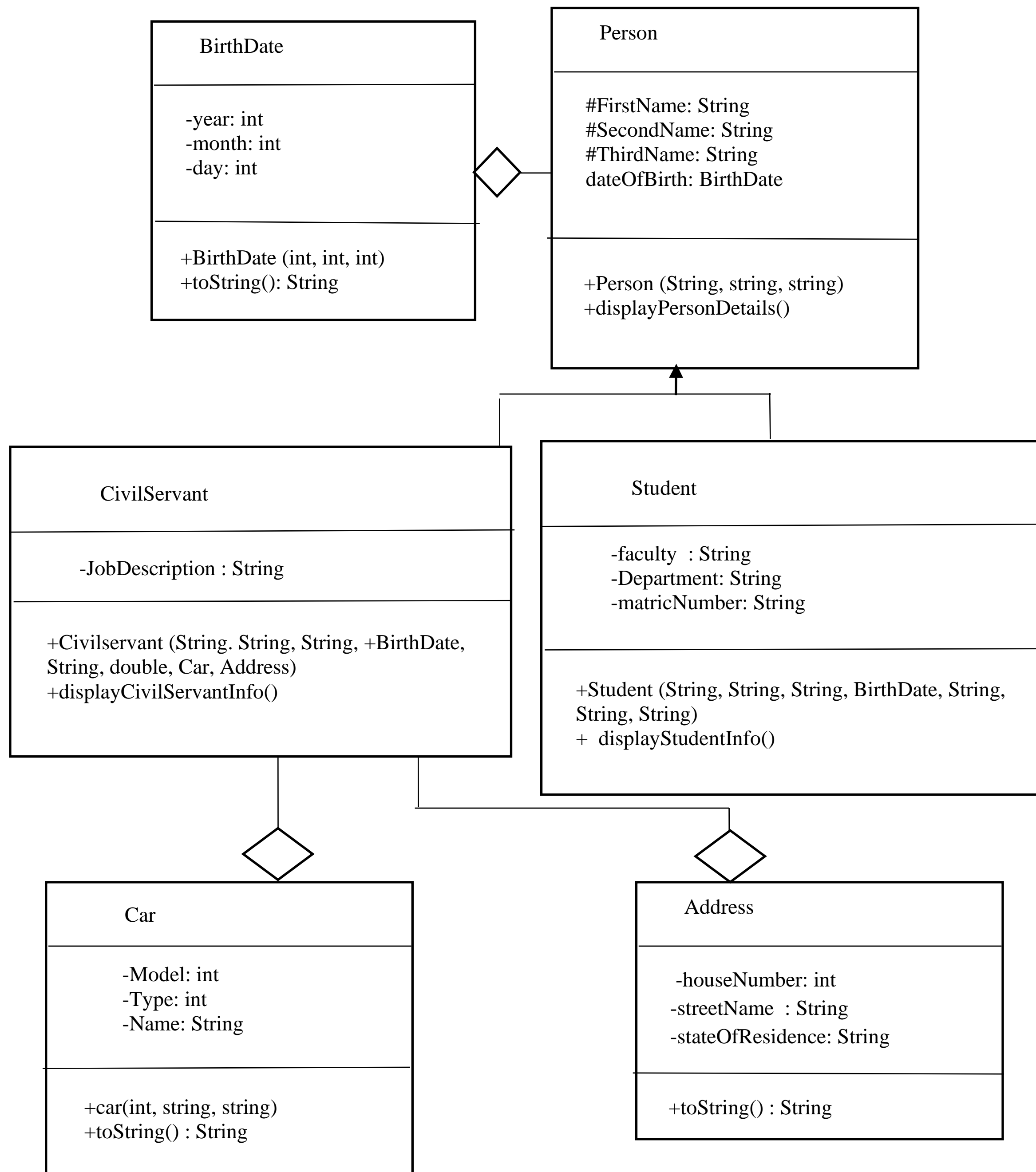


Figure 1: Design of the problem based on the methodology.

RESULT AND DISCUSSION

IMPLEMENTATION

```
package tutorials;
public class Person {
        protected String FirstName;
        protected String SecondName;
        protected String ThirdName;
        protected BirthDate dateOfBirth;

                public Person (String FirstName, String SecondName, String ThirdName, BirthDate dateOfBirth )
                {
                        this.FirstName = FirstName;
                        this.SecondName = SecondName;
                        this.ThirdName = ThirdName;
```

```
                this.dateOfBirth= dateOfBirth;
            }
            public void displayPersonalDetails(){
                System.out.println("first name : " + FirstName + "\n" + "Second name : " + SecondName +
                "\n" + "Third name : " + ThirdName );
                }
}

package tutorials;
public class BirthDate
{
        private int year;
        private int month;
        private int day;
        public BirthDate (int year, int month, int day){
        this.year = year;
        this.month = month;
        this.day = day;
        }
            public int getYear(){
            return year;
            }
            public int getMonth(){
            return month;
            }
            public int getDay(){
            return day;
            }
            public String toString()
            {
             return "year of birth: " + year  + '\n'  +  " Month : " + month + '\n' +  " Day:" + day;
            }
        }
}

package tutorials;
public class CivilServant extends Person
{
        private String Jobdescription;
        private double salaryAmount;
        private Car MyCar;
        private Address Residence;
        public CivilServant (String FirstName, String SecondName, String ThirdName, BirthDate dateOfBirth, String
        Jobdescription, double salaryAmount, Car MyCar, Address Residence){
        super (FirstName, SecondName, ThirdName, dateOfBirth);
        this.Jobdescription = Jobdescription;
        this.salaryAmount = salaryAmount;
        this.MyCar = MyCar;
        this.Residence = Residence;
}
        public void displayCivilServantInfo(){
        System.out.print("Designation : " + Jobdescription + '\n' + "salary Amount: " + salaryAmount + '\n' );
        }
        }
package tutorials;
public class Car {
        private String model;
        private String CarName;
        private int YearMade;
        public Car(String model, String CarName, int YearMade) {
            this.model = model;
            this.CarName = CarName;
            this.YearMade = YearMade;
            }
            public String getModel(){
            return model;
            }
```

```
                    public String getCarName(){
                    return CarName;
                    }
                    public int getYearMade(){
                    return YearMade;
                    }
                    public String toString(){
                    return " model  : " + model + '\n'   + "Car Name : " +  CarName + '\n' + "Year made :" + YearMade;
                    }
          }
package tutorials;
public class Student extends Person {
          private String Faculty;
          private  String department;
          private  String MatricNumber;
                    public Student (String FirstName, String SecondName, String ThirdName, BirthDate dateOfBirth,
                    String Faculty,  String department, String MatricNumber){
                              super (FirstName, SecondName, ThirdName, dateOfBirth);
                              this.Faculty = Faculty;
                              this.department = department;
                              this.MatricNumber = MatricNumber;
                              }
                              public void displayStudentInfo(){
                              System.out.print("Faculty: " + Faculty + '\n' + "Department: " + department + '\n'  + "Matric
                              Number : " + MatricNumber);
                              }
                              }
package tutorials;
public class TestDriver  {
public static void main(String[] args) {
Address address = new Address (2, "Pos-tgraduate School Way", "Kaduna");
Car car = new Car ("Toyota ", "Camry", 2020);
BirthDate  worker = new BirthDate(1985,06, 28);
BirthDate  student = new BirthDate(2005,11, 30);
Student s = new Student ("Godwin", "Akong", "Otu", student, "Science", "Physics", "U19200153");
CivilServant c = new CivilServant("Raphael", "Ushiekpan", "Ugbe", worker,"Lecturer", 200000, car,address);
System.out.println("PROGRAM OUTPUT");
System.out.println(" CIVIL SERVANT INFORMATION");
c.displayPersonalDetails();
c.displayCivilServantInfo();
System.out.println(worker);
System.out.println(address);
System.out.print(car);
System.out.println();
System.out.println("STUDENT INFORMATION");
s.displayPersonalDetails();
s.displayStudentInfo();
System.out.println();
System.out.println(student);

}

}
```

PROGRAM OUTPUT

```
CIVIL SERVANT INFORMATION
first name:        Raphael
Second name:       Ushiekpan
Third name:        Ugbe
Designation:       Lecturer
salary Amount:     200000.0
year of birth:     1985
Month:             6
 Day:              28
House Number:      2
Street Name:       Postgraduate School Way
State:             Kaduna
```

model: Toyota
Car Name: Camry
Year made: 2020

STUDENT INFORMATION
first name: Godwin
Second name: Akong
Third name: Otu
Faculty: Science
Department: Physics
Matric Number: U19200153
year of birth: 2005
Month: 11
Day: 30

The results clearly illustrate how software is elegantly enhanced using aggregation and inheritance. Considering many researches in the field of software development using object-oriented approach, this research presents a comprehensive and concise procedure on program development process. The research did not delve into explaining the details of these two pillars of programming using objects, but describes in detail together with the implementation on when and how to apply the concept of inheritance and aggregation. Incremental program development has been discussed in some researches but application of this residual knowledge is what this research dwell on (Noble and Potter, 1997).
The result of the research also shows that software can easily be reused and adapted to solve other problems in similar or varied domains. For example, the address and personal detail classes can be used in other programs just like we have used for the student and lecturer class in the research.
The program output is also presented in a simple order, so that the working of the implementation can be understood from the implementation, likewise adjustment can be made in the implementation to give a different formatted output.

CONCLUSION

The research clearly describes explicitly the concept of aggregation and inheritance, by applying the similarity, disparity and whole-part methodology. Together with the design and implementation these concepts has now been elaborately discussed so that they can be applied appropriately during problem solving. From the methodology it is identified that it is easier to enhance a code with aggregation than inheritance. But these important concepts complement each other, so, both are mutually exclusive but, in all inheritance, should only be used when it will not be efficient to used aggregation.

REFERENCES

Asagba, P., Ogheneovo, E. (2010). A Comparative Analysis of Structured and Object-Oriented Programming Methods. Journal of Applied Sciences and Environmental Management (ISSN: 1119-8362) Vol 12 Num 4. 11. 10.4314/jasem.v11i4.55190.

Hutabarat, B. I., Purnama, K, E., Hariadi M. (2009). The 5th International Conference on Information & Communication Technology and Systems

Klump, R.P. (2001). Understanding object-oriented programming concepts. *2001 Power Engineering Society Summer Meeting. Conference Proceedings (Cat. No.01CH37262), 2*, 1070-1074 vol.2.

Noble, J., Potter, J. (1997). Change Detection for Aggregate Objects with Aliasing. Conference Paper · January 1997 DOI: 10.1109/ASWEC.1997.623759 · Source: IEEE Xplore

Otu, G. A., Achimugu, P., Owolabi, A., Raphael, U. U., Abdullahi, M. J., Shukurah, O. M., Blamah, N. V., & Usman, S. L. (2022). TRANSPORT SERVICE SYSTEM DESIGN USING MODIFIED APRIORI ALGORITHM. *FUDMA JOURNAL OF SCIENCES*, *6*(4), 25 - 36. https://doi.org/10.33003/fjs-2022-0604-845

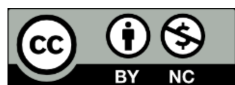
Prehofer, C. (2001) Feature-oriented programming: A new way of object composition. Concurrency and Computation: Practice and Experience Concurrency; 13:465–501 (DOI: 10.1002/cpe.583).

Taivalsaari, A. (1996) On the Notion of Inheritance Nokia Research Center. ACM Computing Surveys, Vol. 28, No. 3.
Tung, S.S. (1992). Interactive modular programming in Scheme. *LFP '92*

Vincent O. R., Afolorunso A. A. (2020) CIT383 Introduction to Object-Oriented Programming. National Open University of Nigeria

Weixin Z, Sun Y., Oliveira-Bruno, C. d. S. (2021). Compositional Programming. ACM Trans. Program. Lang. Syst. 43, 3, Article 9, 61 pages. https://doi.org/10.1145/3460228

Zeynab, R. (2015). Properties of Relationships Among Objects in Object-Oriented Software Design. International Journal of Programming Languages and Applications. 5. 10.5121/ijpla.2015.5401



©2023 This is an Open Access article distributed under the terms of the Creative Commons Attribution 4.0 International license viewed via https://creativecommons.org/licenses/by/4.0/ which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is cited appropriately.