# A Genetics-Based approach to inheritance modeling

By

Dareen Mousa Mohammad Hamoudeh

Supervisor

Prof. Said Ghoul

This Thesis was Submitted in Partial Fulfillment of the Requirements for the Master's Degree in Computer Science

Deanship of Academic Research and Graduate Studies

Philadelphia University

April 2012

## جامعة فيلادلفيا
## نموذج تفويض

أنا دارين موسى محمد حمودة ، أفوض جامعة فيلادلفيا بتزويد نسخ من رسالتي للمكتبات أو المؤسسات أو الهيئات أو الأشخاص عند طلبها.

التوقيع :
التاريخ :

## Philadelphia University
## Authorization Form

I am, Dareen Mousa Mohammad Hamoudeh, authorize Philadelphia University to supply copies of my thesis to libraries or establishments or individuals upon request.

Signature:
Date:

# A Genetics-Based approach to inheritance modeling

By

Dareen Mousa Mohammad Hamoudeh

Supervisor

Prof. Said Ghoul

This Thesis was Submitted in Partial Fulfillment of the Requirements for the Master's Degree in Computer Science

Deanship of Academic Research and Graduate Studies

Philadelphia University

April 2012

Successfully defended and approved on _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _

**Examination Committee Signature** **Signature**

Dr. , Chairman. _ _ _ _ _ _ _ _ _ _ _ _ _
Academic Rank:

Dr. , Member. _ _ _ _ _ _ _ _ _ _ _ _ _
Academic Rank:

Dr. , Member. _ _ _ _ _ _ _ _ _ _ _ _ _
Academic Rank:

Dr. , External Member. _ _ _ _ _ _ _ _ _ _ _ _ _
Academic Rank:

# Dedication

To who helped me and supported me along my way to success; my mother, my father, my sisters and brothers. I humbly dedicate this work for them, with all my sincere gratitude.

**Dareen Hamoudeh**

## Acknowledgment

I would like to extend my thanks and sincere gratitude for who has guided me through my study and my thesis work; prof. Saed Ghoul, and all my teachers.

Also, I am grateful for those who supported me and encouraged me in any way; my family, my friends and my superiors and colleagues at work

**Dareen Hamoudeh**

# Table of Content

| Subject | Page |
|---|---|
| Authorization Form | i |
| Title | ii |
| Examination Committee | iii |
| Dedication | iv |
| Acknowledgement | v |
| Table of Contents | vi |
| List of Tables | viii |
| List of Figures | ix |
| Abstract | x |
| Chapter One: Introduction | 1 |
| 1.1 Problem Statement | 2 |
| 1.2 Motivation | 6 |
| 1.3 Contribution | 6 |
| Chapter Two: Case Study | 9 |
| Chapter Three: Selective Inheritance Approaches | 14 |
| 3.1 Selective Inheritance at Class Level | 15 |
| 3.2 Selective Inheritance at Object Level | 17 |
| Non-genetic approaches | 18 |
| Genetic approaches | 19 |
| Chapter Four: A Genetics-Based Approach to Inheritance Modeling | 24 |
| 4.1 Some New Definitions | 25 |

| | |
|---|---|
| 4.2 Each Artificial Entity is a Phenotype Developed from a Genotype of a Genome | 29 |
| Genome | 29 |
| Genotype | 34 |
| Phenotype | 35 |
| 4.3 The Nature of a Genotype is completely operative | 36 |
| 4.4 Applying into our Case Study | 44 |
| Chapter Five: "Is-a" Vs. "Composed by" Inheritance Model and Complexity | 49 |
| 5.1. "Is-a" Vs. “Composed by” Inheritance Models | 50 |
| 5.2 Complexity Measurements | 52 |
| Inheritance hierarchy | 52 |
| Nested classes | 55 |
| 5.3. Combining between our Approach and the "is-a" Hierarchy Model | 57 |
| Chapter Six: Implementation Issues, Evaluation and Application Areas | 59 |
| 6.1 Implementation Issues | 60 |
| 6.2 Evaluation Criteria | 61 |
| 6.3 Application Areas | 63 |
| Chapter Seven: Conclusion and Future Works | 65 |
| 7.1 Conclusion | 66 |
| 7.2 Future Works | 66 |
| References | 67 |
| Appendix | 70 |

# List of Tables

| Table Number | Table Title | Page |
|---|---|---|
| Table (4-1) | Comparison between Some Conventional OO and Genetics-Based Concepts. | 26 |
| Table (6-1) | comparison between our approach and the previous approaches. | 63 |

# List of Figures

| Figure Number | Figure Title | Page |
|---|---|---|
| Figure (1-1) | Dolphin inherits selectively from Fish and Mammals classes | 3 |
| Figure (1-2) | Natural Inheritance Process in Flowers | 7 |
| Figure (2-1) | Shapes super-class and its sub-classes | 11 |
| Figure (2-2) | Using Specialization to Implement Methods in Subclasses | 12 |
| Figure (2-3) | sample on shapes hierarchy that may be created for Geometry | 13 |
| Figure (3-1) | The Use of Selective Inheritance in Martin Work, Using its own Notations | 16 |
| Figure (3-2) | Applying *Dori* Approach on our Case Study, Using its own Notations. | 17 |
| Figure (3-3) | Applying Al-Ahmad's Approach on our Case Study | 18 |
| Figure (3-4) | Applying Semantic Classification on our Case Study | 20 |
| Figure (4-1) | Inheritance Definition in Conventional OO and in Genetics-Based Model | 28 |
| Figure (4-2) | Genome Model, Versions of Characteristics. | 29 |
| Figure (4-3) | Genome Structure. | 33 |
| Figure (4-4) | Genome, Genotype and Phenotype | 35 |
| Figure (4-5) | Genotype | 36 |
| Figure (4-6) | Interpretation process of the Genotype Program. | 41 |
| Figure (4-7) | Shapes inheriting using our approach. | 46 |
| Figure (5-1) | Inner Class Complexity from the Perspective of Breadth and Depth | 56 |
| Figure (5-2) | Figure (5-2): Deducting a Portion from Figure (4-6) that is Relevant to the Portion in the Case Study. | 57 |
| Figure (5-3) | Combining "is-a" Model with "Composed by" Model. | 58 |

# Abstract

The conventional Inheritance concept which is adopted in the current Object Oriented Programming (OOP), where it is acting on "is-a" hierarchy model, has some defects. Where OOP is trying to be more close to the real life, it is still far from genetics principles.

Inheritance means that the child class can inherit, and get everything that is public in the parent class automatically. This process has solved many problems, but it does not simulate what is really happening in our life, where each object can gain just the needed traits from the parent class. That means the conventional inheritance is not selective and is generating identical objects.

While conventional Inheritance acts on “is-a” hierarchy model, the works that introduced the selective inheritance were also done on this model.

The inspiration from genetics has led to a selective inheritance acting upon a “Composed by” model rather than the “is-a” model; where in biology, the needed traits are selected from the Genome which holds all the aspect traits. Those traits are classified into several classes according to "Composed by" relation.

After evaluation, it has shown that the selective inheritance that acts on "composed by" model is better than the selective inheritance over “is-a” hierarchy model.

# CHAPTER ONE

# INTRODUCTION

Since Object Oriented Programming has been created, in which everything in a program is an object, which tries to mimic real life, a lot of concepts have been adopted, and one of the most important is the inheritance.

Inheritance means that the child class can inherit, and get everything that is *public* in the parent class automatically. This process minimizes the amount of duplicate code in an application by putting common code in a parent class and sharing it amongst several child classes. This process is applied on a "is-a" hierarchy model.

## 1.1 Problem Statement

Although this concept solved many issues, there also were some defects. Existing Inheritance (classical) obligates objects to have the same behavior as the parent class (ancestor) because when they inherit from a super-class , they get all public attributes and behaviors from that super-class (even if we do not want some of those "things")(A.Pillay,2007), and latter each object can add additional behavior to provide special action for specific needs. By this way, we can have many identical objects in the same class. But this does not happen in real life. Taking into consideration that we want to be as close as possible to real life, where objects can inherit selectively specific and needed traits (variables/methods) from one or more class as long as they are compatible, so that we can have different objects for each class.

Selective inheritance (S.Brinke, 2007, J.Bastian et al, 2005, S.Ghoul, 2011) offers many of the benefits of multiple inheritance and avoids name overlapping and repeated inheritance problems, and it limits the amount of information in each object, so we can focus our attention on the features which are relevant to our interest.

Assume that we have two Classes: Fish class and Mammals class, where:

```
Class Fish
 {
    Live-In-Water ();
    Swim ();
    Lay-Egg ();
    Has-Fins ();
    Breath-By-Gills ();
 }
```

```
Class Mammals
 {
    Live-On-Land ();
    Walk ();
    Give-birth ();
    Has-legs ();
    Breath-by-lungs ();
    Has-sound () ;   }
```

If we want to create a class *Dolphin*; which class should it inherit from?

By applying current OOP inheritance concepts, if Dolphin inherits from Fish and Mammals that means Dolphin will lay-egg () and give-birth () at the same time, or we should create a new class called Dolphin to place the required traits, but, why can't Dolphin select and inherit just the needed traits (methods/ variables) from Fish and Mammals as shown in Figure (1-1).

By achieving this feature we can minimize the number of classes, by allowing the class to select and inherit specific and required traits from existing classes instead of building a new class that re-contains traits existing in other classes.

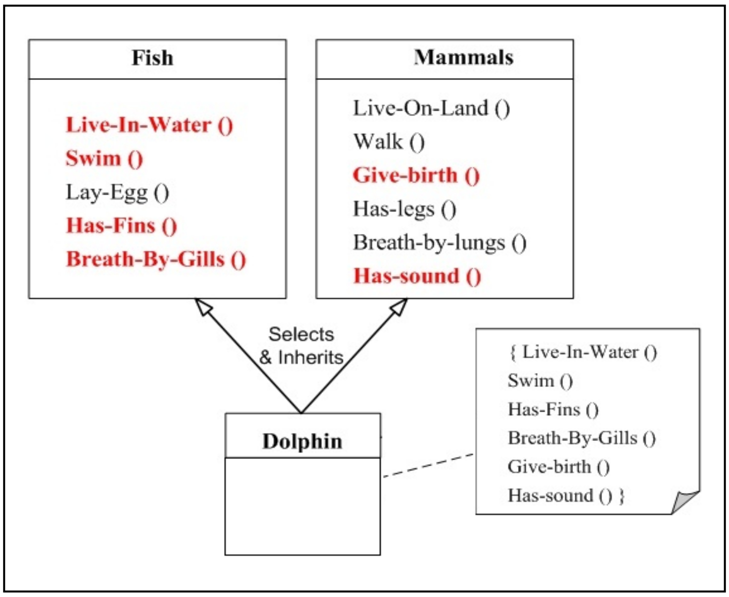


Figure (1-1): Dolphin inherits selectively from Fish and Mammals classes

Several works have agreed on the importance of using selective inheritance concept (T.Oplustil, 2002, S.Brinke, 2007), and others have proposed the selective inheritance as a solution to use in their works (J.Bastian et al, 2005, N.Sakkinen, 2005, S.Herrmann, 2005).

To solve the overlapping properties in classical inheritance, semi-selective inheritance mechanism was suggested, and a simple syntax for it was proposed where the object can only select parts of what it inherits (T.Oplustil,2002),.

In Intel IA-32 Specification (J.Bastian et al, 2005), the term conditional inheritance was used where a parent with the best fit is pre-determined to the object, not the object who selects the best fit. This will not make the object to be flexible and free to choose traits implicitly as we aspire.

Reverse inheritance, which is not supported in the current OOPLs, was proposed in (N.Sakkinen, 2005) as generalization inheritance. Author argues that it is preferable to be selective, because not all common features need to be exherited (generalized from subclasses to super-classes).

Another case and different approach of using selective inheritance was introduced in the dynamic view of methods in role class, a callout binding approach is used to bind methods from a role class to its base (S.Herrmann, 2005). The idea of this binding is to apply a selective inheritance, where mapping a feature in a callout means that it is shared between the role and the base, otherwise, it is invisible at the role.

Due to that importance, and that object oriented programming languages do not support the selective inheritance feature (S.Brinke, 2007), several studies have been done on Selective inheritance, the majority relied on the use of genetics concepts, but two different approaches were adopted: Selective inheritance at class level and selective inheritance at object level.

Selective inheritance at class was proposed in (T.Martin, 2004), which means, it is not the object which selects the desired traits, but the class restricts the traits which are not desired to be inherited by specifying whether the trait is inheritable or not. Here, the object will remains forced to inherit all the inheritable traits which mean that the identical objects problem has not been solved and the object does not have the permission to inherit only what it needs.

Selective inheritance and genetics principle have been used also in the multiple inheritance field (D.Dori, 1994), where instead of inheriting all features from all ancestors, the object will dynamically select any ancestor subset to inherit from. This approach is not working on feature selection, as the object will select its ancestors but it will inherit all their features.

Selective inheritance at object level is the closest to the real life, where the object is more flexible, able and allowed to select/ reject required but not contradictory traits from the ancestor(s) inheritable traits. The explicit and coherent selection or rejection of any property in the class hierarchy is done according to control rules (J.Meslati and Ghoul, 1997, S.Ghoul. 2011).

Non-genetic approach was adopted by (Al-ahmed et al, 1999). Al-ahmad has addressed the issues related to instance variables and methods in specialization inheritance, and proposed solutions to inherit with maximum code reuse and minimum operation redefinition. The proposal has an efficiency problem and it needs support from the compiler.

Genetic approaches were adopted in several studies. (J.Meslati and Ghoul, 1997) introduced the concept of semantic classification. The work proposed the alternation concept and a genetic program associated with each class to constitute variants of class. This work uses the concepts of conventional classes, and the object may inherit from several classes.

The work (S.Ghoul, 2010) has proposed a model that suggests a platform for genome evolution and genotype definition process. Where the author has clarified the fundamental principles of the integrated model for bio-inspired systems, he demonstrated the relation between the genome and the genotype (Selected traits) where the latter is a genome with Enabled/Disabled traits. These traits are defined according to control rules.

In (S.Ghoul, 2011), the work has introduced several concepts that were not supported by current Aspect-Oriented paradigm. The author has also provided a coherent software design methodology that combines the Bio-Inspired approach together with the Object-Oriented, Aspect-Oriented, and Subject-Oriented approaches. The work has suggested an implementation for the Genome configuration (Coherent selected variations), but without modeling the relations between the genetic concepts; genome, genotype and phenotype, and without studying the control genes/ rules that govern the classification inside the genome and those which control the selection process.

## 1.2 Motivation

From the previous studies, we state that several insufficiencies exist:

- Current OOPL inheritance is acting in “is-a” hierarchy, this leads to several problems (exponentially increased ancestors, conflicts in attributes name …etc.).
- Same objects of the same class ("whole" conventional inheritance or selective inheritance on class level).
- Several classes exist to allow objects to collect their properties (Selective inheritance on object level).
- Objects are complex and have unwanted properties that never used.
- The actual bio-inspired approaches are general and not really modeled.

Recently, the importance of using bio-inspired (J.Blaylock, 2008, S.Ghoul, 2011, W.Braynen et al, 2007 and T.Otter, 2005) concepts began to spread and has led all works to be tended to it.

## 1.3 Contribution

So, inspired from genetics, our work aims to deal with the previous problems by:

- Replacing the “is-a” inheritance hierarchy model used in current OOPL by the “composed by” inheritance model induced by the Genome architecture; where there will only be one class “composed by other classes” that holds all properties associated to specific aspect instead of inheriting these properties from several classes and their ancestors.
- Formally and deeply model the selective inheritance based on the “composed by” class model.
- Studying the value of “composed by” selective inheritance model relative to the “is-a” selective inheritance model and the composition of the two models.
- Use and develop the language extension proposed in (S.Ghoul, 2011) for the current OOPL to be able to implement our approach.

This will lead to:

- Reducing the number of classes.
- Allowing different objects to be instantiated from the same class, which is a genetic class: the Genome.

- Discharging the object from not useful properties.

In Figure (1-1) we illustrated a very simple example to explain the actual general idea of selective inheritance. In Figure (1-2) we illustrate the selective inheritance using our approach. The figure shows a natural implementation in the flowers which are seen in many types and colors, where each flower inherits its traits from a "*Genome*" class called Flowers. The Flowers Genome is composed of several traits"*Genes*": Colors, Petals, etc. Each trait in turn contains a set of alternatives "*Allele*". One or more alternatives are selected from each trait; selecting a specific color from all colors alternatives, a specific petal shape from all petals alternatives, etc., but the selection process depends on several rules and controls and may be affected by other factors. But the "final gained traits" must be selected in a manner that ensures their compatibility.

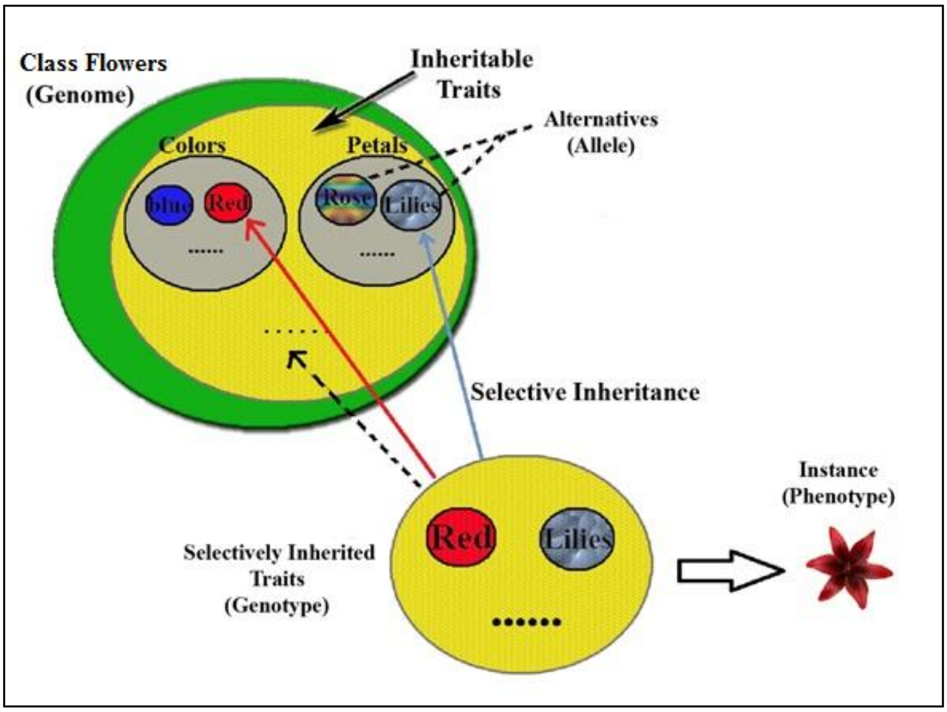


Figure (1-2): Natural Inheritance Process in Flowers

Selective inheritance is one of the promising concepts in OOP. It can be represented by several approaches, but in our work, we aim to extend the benefits by getting closer to the real world and inspiring from the genetics concepts in modeling our model.

So, our selective inheritance modeling is specified by:

- Genotype Program, to define wanted traits or the most important ones to be enabled.
- Genotype Program Interpretation process that is governed by several rules to produce the genotype with wanted and coherent properties.

Our approach is at a conceptual level, we present as formally as it is required for its understanding. Its formalization is out of scope of our work. It may be developed when the idea is largely accepted.

In the following chapter, we will present the Geometric Shapes inheritance as a case study for the whole work. In chapter three, we will give an overview on selective inheritance approaches. In chapter four, we will present our genetic approach to inheritance modeling. In chapter five, we will show its scientific value and complexity. At last, in chapter six, we will present implementation issues, evaluation, and future work. We would like to point out that, all the figures will be presented using UML notation except those which we clarified their legends.

# CHAPTER TWO

# A CASE STUDY

In this chapter, we will introduce a case study which will be used as a support to all our work. Our case study is to illustrate the idea of our approach and not to compute its value.

In Geometry, Shapes are mainly used. These shapes are classified into several categories (polygonal shapes like square, curved shapes like circle, etc.); also shapes can combine between two categories (such as semicircle, crescent, etc.). Any shape category can also be either flat or solid (balls, cubes, cones, etc.).

Each shape has its own properties (sides, angles, center, etc.) and its special functions (area, perimeter, etc.). The function for a particular shape has its own body that differs from other shapes function although they may have the same name; e.g. the area of the circle is different from the area of the square.

To define any of these shapes using OOP, we must choose the correct class to inherit the appropriate properties and methods for that shape.

So, if we want a *Rectangle*, we assume that:

-We have a class called *Polygons*:

```
Class Polygons
  {
     Length, Width, X, Y; // X,Y for center.
     Angle1, Angle2, Angle3, Angle4;
     Area ();
     Circumference ();
     Center ();
  };
```

-Then, a class *Rectangle* can inherit from the *Polygons* class, as follows:

```
Class Rectangle: public Polygons
  {
     Diagonal ();
     ….. ;
  }
```

After that, in the main class, we instantiate *rectangle1, rectangle2* objects from the *Rectangle* class, as follows:

```
Rectangle rectangle1, rectangle2;
```

As we notice, *Rectangle* class has inherited all public properties from *Polygons* class and it can add new properties (such as Diagonal (), etc.).

*rectangle1 and rectangle2* will have all public properties that exist in *Polygons* class and those which exist in *Rectangle* class. *rectangle1* and *rectangle2* are similar objects in terms of that they have the same properties and methods.

Now, if we have been asked to create a *circle* Object, we will not be able to use *the Polygons* class, so we will assume that:

- We have class called *Curved*:

```
Class Curved
 {
     Radius, X, Y; // X,Y for center.
     Area ();
     Circumference ();
     Center ();
 }
```

-And a class *Circle* that inherits from it:

```
Class Circle: public Curved
 {
     Move ();
     … ;
 }
```

As we notice, there are similar properties in *Curved* and *Polygons* classes. So, to apply the *reuse* feature, we can merge repeated properties into one super-class, let it be *Shapes* class, and then the two classes *Curved* and *Polygons* can be sub-classes from it, as in the figure (2-1). After that, by using *Extension* concept each subclass can add new properties and methods.

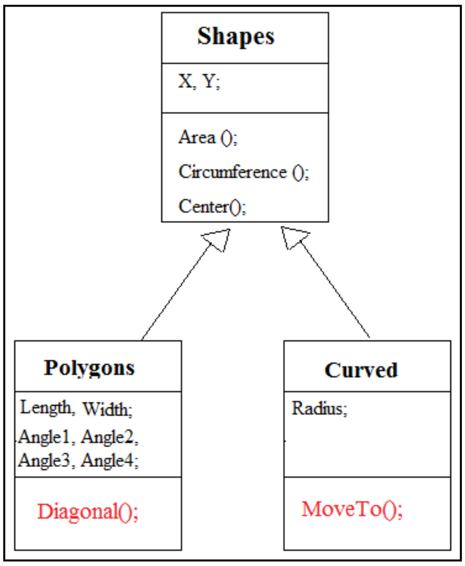


Figure (2-1): *Shapes* super-class and its sub-classes

Because *area ()* and *Circumference* () for *Circle* class are different from *area ()* and *Circumference* () for *Rectangle* class; they are not implemented in *Shapes* class and left for the child classes to implement them, this is known as *specialization* as shown in figure (2-2).

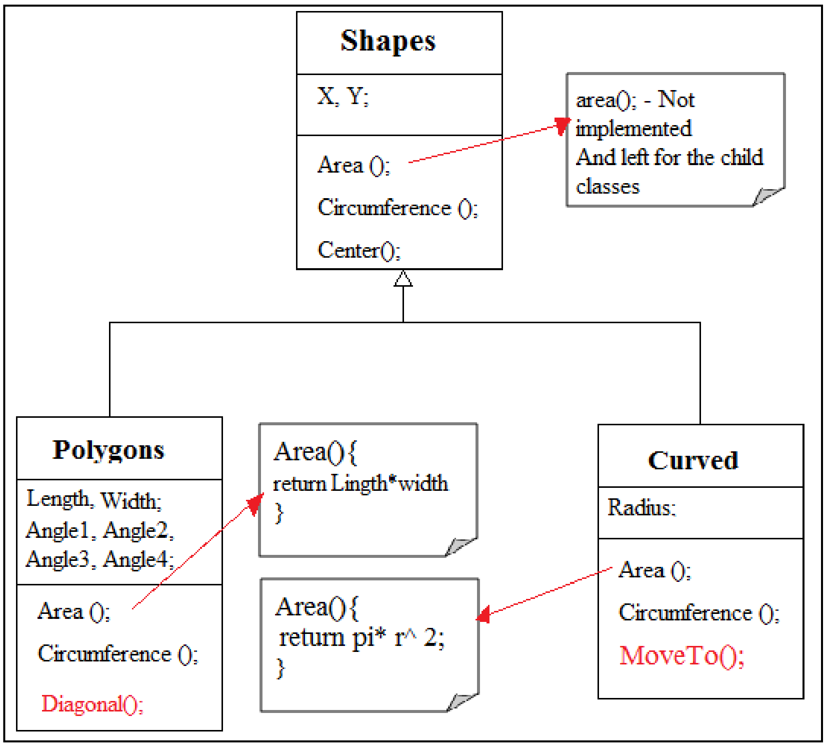


Figure (2-2): Using Specialization to Implement Methods in Subclasses

Now, let’s create a *Cone* object. It needs different properties, it consists of a circle and a pyramid, but as we see these properties exist in the two classes *Curved* and *Polygons*. *Cone* class can inherit its properties from both classes, but it's still missing an important one which is the height which makes it a *stereophonic*. So it will add this new property. Also, if we want to create a *Cube* object, it will inherit from *Polygons* class and will add the height property. From that we conclude that a new class called *Stereophonic* must be created as a super class for *Cube* and *Cone* classes, But, what about *Balls* and *Cylinders* classes?

With the growing need for new shapes and properties, new classes will be created, which makes the hierarchy more complicated, and the objects more complex; for example, what if *circle1* object does not want the method MoveTo () which exists in the *Curved* class?

Other types of shapes make the previous classification incomplete, where, *Quinary* is a shape that consists of five sides, two of them are parallel, while the rest are not, and it contains five angles. Assume that we have a class for each kind of polygons, how many one will we create?

Figure (2-3) shows one simple sample for the shapes hierarchy that may be created, if we also take into account that there is *Parallelogram*, *non- Parallelogram* and *Flat* shapes.

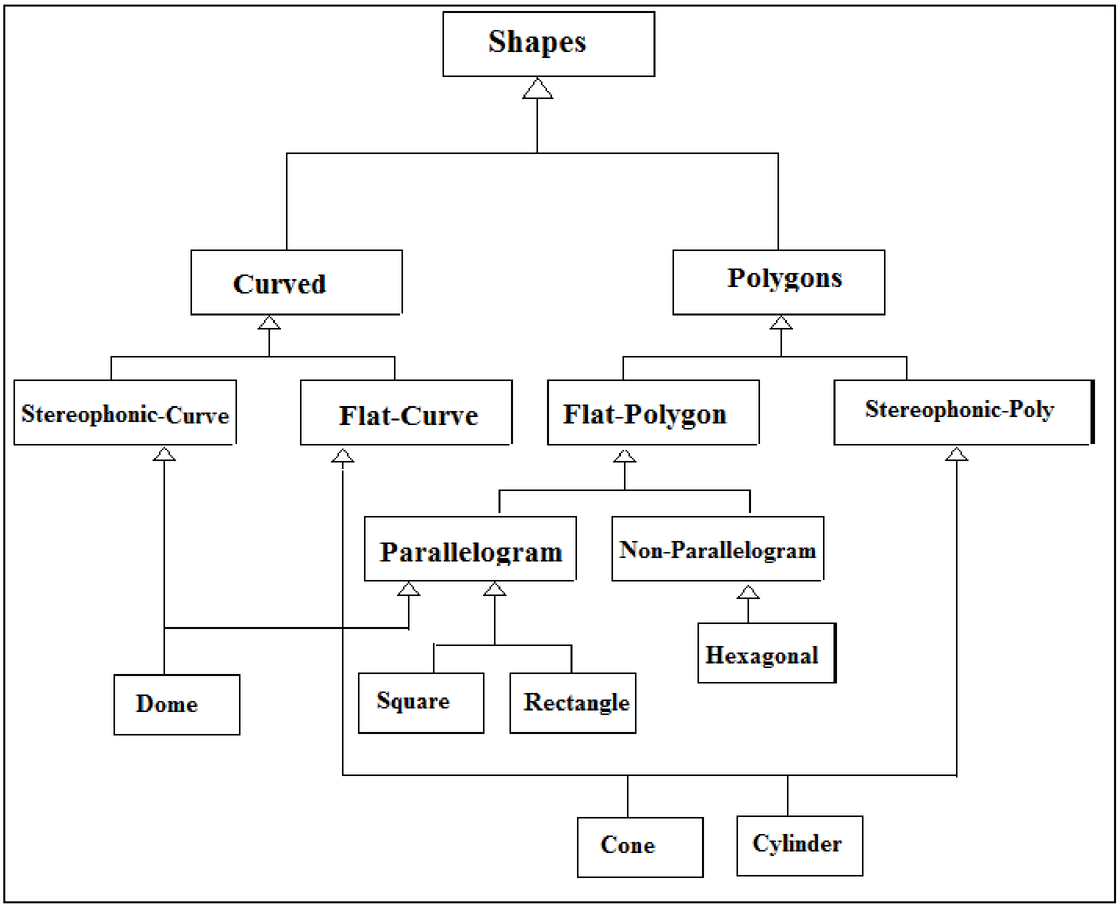


Figure (2-3): sample on shapes hierarchy that may be created for Geometry.

# CHAPTER THREE

# SELECTIVE INHERITANCE APPROACHES

Because of the importance and the growing need for selective inheritance, there are a lot of studies in this area. Approaches are different and the topic has been analyzed from different aspects. But they always worked on the current inheritance "is-a" hierarchy model

As is known, inheritance is a process that a child class gains its properties from a parent class. For that, some approaches have adopted the selection process at the class level, where the others have done it at the object level. In the following we will consider these two approaches.

## 3.1 Selective Inheritance at Class Level

(T.Martin, 2004) addressed some problems associated with computer languages, and he concludes that there must exist a more useful set of semantic entities which is capable of providing greater expressiveness and intuitiveness than is available from conventional computer languages. He provided a program and a method of managing entities in an Object Oriented environment in which parameters are selectively inherited from the parent into child responsive to persistent indications of the inheritability of these parameters stored in a non-volatile memory.

Selective inheritance feature adopted in this work is that it does not require changes to an underlying database schema. Parameters in a parent entity, which is not desired to be inherited by a child entity, may be selectively restricted from being inherited. So, fields which are capable to be inherited are referred to as "gene" fields, while others are referred to as "non-gene". This approach requires that the class is who determines which traits to be allowed\prevented to be inherited by a child. This will make the object confined to set of traits to inherit them; this will still lead to identical objects from the same class. Figure (3-1) shows how to use selective inheritance, in our case study, using this approach.

By the approach, a *Shapes, Polygons* and *Curved* entities have been modeled. All three entities are assigned different types of *Areas* () and *Circumference* () methods. To avoid storing an empty or unused field on *Curved* or *Polygons,* the *Shapes' Area* () and *Circumference* () fields may be defined as *non-gene,* while other fields *X, Y* and *Center* () that are common to a *Shapes, Curved* and *Polygons* entities may be defined as *genes,* and thus inherited into *Curved* and *Polygons.* Also, *Curved* and *Polygons* may have an *Area* () and *Circumference* () fields that may be a gene or non-gene. And

*Shape's* non-gene fields are kept from being inherited into the *Curved* and *Polygons*. Through this selective inheritance concept, we avoid any programming errors or confusion that may happen when an entity has two *Area* () or *Circumference ()*.

*We conclude that,* the parent (class) in this approach holds two types of parameters: inheritable and non-inheritable, where all the inheritable parameters will be inherited into child (object). Therefore the object does not have the ability to select only the parameters that it needs.

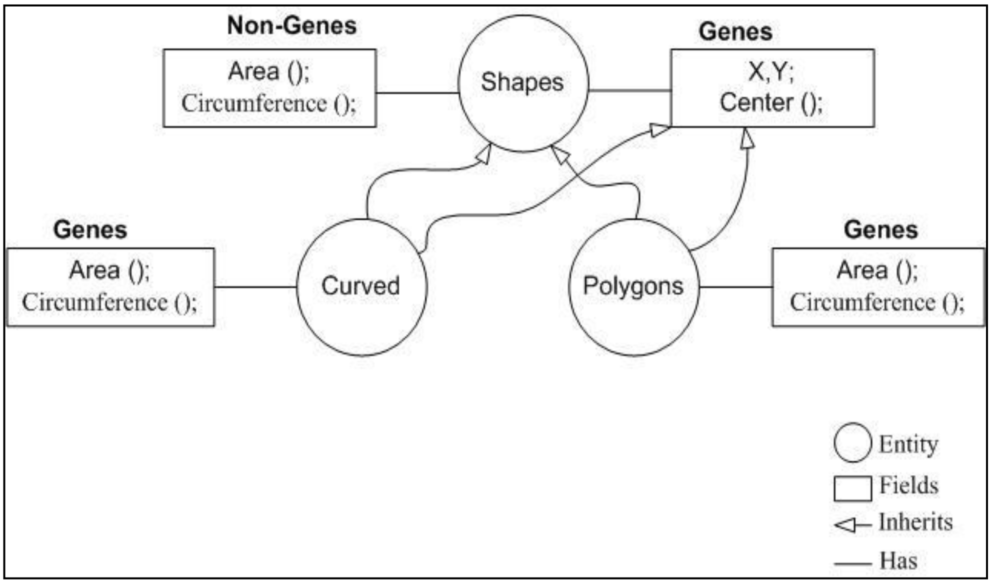


Figure (3-1): The Use of Selective Inheritance in (T.Martin, 2004) Work, Using its own Notations.

Another work (D.Dori et al, 1994) has adopted the selectivity at class level, but it was in the multiple inheritance field. In conventional multiple inheritance, a subclass (child) inherits from more than one super class (parent), and so, it will inherit from all the super classes ancestors - of course will inherit all the features of the super-classes and ancestor – and there is no option to select specific ancestors. When applying conventional multiple inheritance many problems may occur, including inheriting a feature that is a contradictory with other features, or inheriting repeated features.

In this work, an embryonic class notion was used to develop a generalized approach that allows the class to dynamically select any ancestor subset. The embryonic class contains a default attribute called ancestor-list, which is a list of ancestors from which the class inherits its features. An implicit method "*Formulate"* accepts the ancestor-list as a parameter and constructs the internal structure accordingly. With selective multiple inheritance, a class may inherit features from any number *m* of a given set of ***n*** ancestor classes. Many of this approach ideas were inspired from the inheritance in biological systems where the observed features do not expose the entire genetic information.

This approach is convenient for multi- level multiple inheritance. So, by applying this on the case study that we have, we will be forced to use the special case that the author talked about, which is the “conventional multiple inheritance”; because all the ***n*** ancestors are active, and no selective inheritance is enabled.

So, To model *Cone* class using this approach, we see that, it must inherit from both *Stereophonic-Poly* and *Flat-Curved* classes which they also must inherit from their ancestors, as mentioned in Figure (2-3), the author said that we can use the model shown in Figure (3-2/b) which is the same as conventional multiple inheritance, instead of the model in Figure (3-2/a).

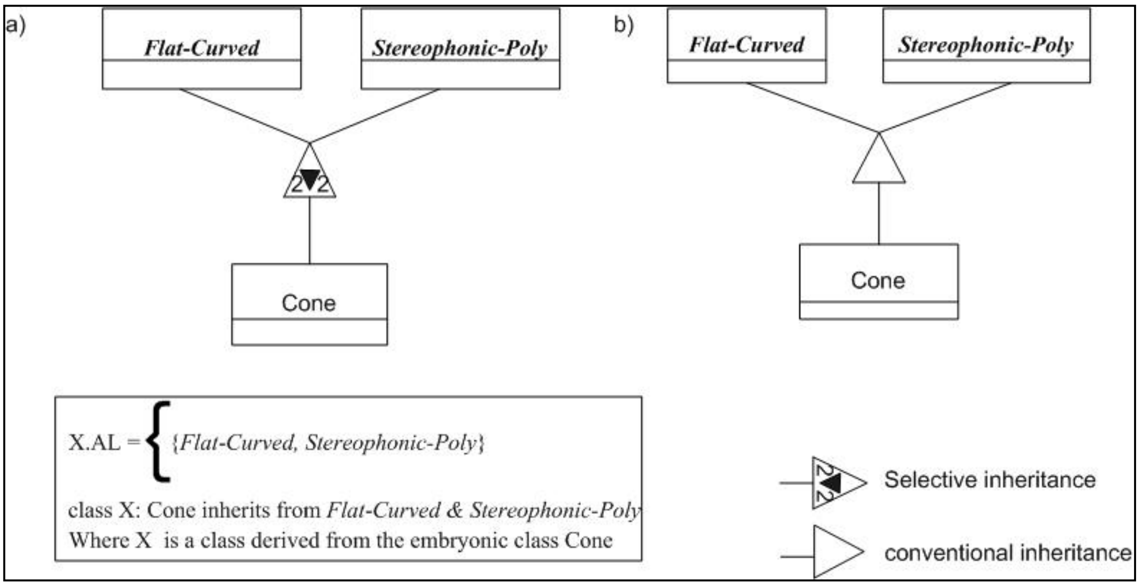

a)
Flat-Curved
Stereophonic-Poly
2
2
Cone
b)
Flat-Curved
Stereophonic-Poly
Cone
X.AL = {Flat-Curved, Stereophonic-Poly}
class X: Cone inherits from Flat-Curved & Stereophonic-Poly
Where X is a class derived from the embryonic class Cone
Selective inheritance
conventional inheritance

Figure (3-2): Applying *(D.Dori, 1994)* Approach on our Case Study, Using its own Notations.

*We conclude that,* this approach is dealing with selecting a set of desired classes not with selecting desired features, beside that, the object is not the one who determines whom to inherit from and even, the object will still forced to inherit all the features from the ancestors' subset. Also, this approach is not convenient in all cases.

## 3.2 Selective Inheritance at Object Level

This approach is the most logical and preferable approach to selective inheritance, where the object can easily select the needed features from any class(s) that holds that features. Therefore, the approach took two trends: genetic and non-genetic approaches.

### Non-genetic approaches

In (Al-Ahmed et al, 1999), the selective inheritance concept was used in a different perspective, where, implementation solutions were proposed to derive a subclass with reducing the redundancy in representation and reducing the number of methods redefinition. These solutions were by using suitable names for the class operations, using inherited names, based on pointer data members, or based on read and write operations. But an adequate solution that was proposed requires the use of new language construct to provide better support. So, the proposed solutions require support from the compiler. Where a new section will be added to the class that lists the data members to be suppressed followed by the relationships between the data members to be used for the generation of correct code for the read and write operations.

Applying this approach on our case study, is to let *square* class be able to inherit from *Rectangle* class, with maximum code reuse and minimum re-definition, although they have a different constraint, where in square length = height, as shown in figure (3-3).

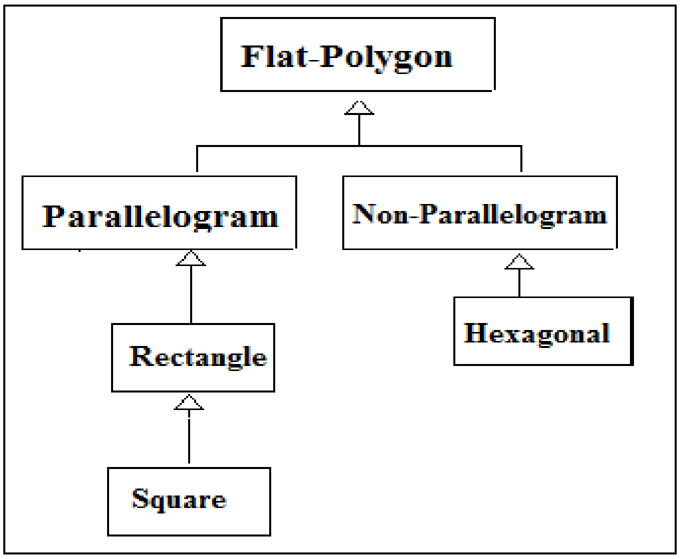


Figure (3-3): Applying (Al-Ahmed et al, 1999) Approach on our Case Study

The Rectangle and Square classes are defined as the following:

```
Class Rectangle
{ // Defineing class Rectangle
Public:
     Rectangle (float len, float wid): length (len), width (wid) {}
     Virtual void NewLength(float len) {}
     Virtual void NewWidth(float wid) {}
     …….
Rename: //an adequate mechanism to re-name the width & length into side.
     Length Side, NewLength NewSide …

Private:
     Virtual Length;
     Virtual Width;
};
// Class Square that inherits from Class Rectangle.
Class Square: Protected Rectangle
{
Public:
     Square (float side) Rectangle (side,side){}
     Rectangle:: NewLength;
…
Protected:
     Void NewDimensions (float len, float wid)
     {
       If (len== wid) WtLength(len); // using a proposed solution based on
                                             //read & write operations.
          Else // error message
     }
Suppress:
     Width< Side = Width, Side>
      // new section added to list data members to be suppressed followed by
     //the relationships between them to be used in correct code generation.
};
```

In conclusion, this study did not meet the real objective of the selective inheritance which is the explicit and coherent selection or rejection of any property in the class hierarchy according to some rules.

## Genetic approaches

These approaches are inspired from biological systems especially the genetic process. Meslati and Ghoul (J.Meslati and S.Ghoul, 1997) have proposed a new approach to classification which they call *semantic classification*. In conventional classification, the class groups objects that have similar structural and behavioural properties. The similarity between objects is based on their syntactic description as well as on their underlying semantics, which include all abstract features beyond physical considerations. This kind of classification is not always convenient where objects that have slightly different profiles must belong to different classes.

By *semantic classification,* authors mean the possibility of making objects which have different profiles (i.e. properties) but identical underlying semantics, instances of the same class. The difference among objects will be achieved by choosing appropriate properties before creating those objects.

Since a semantic class contains all possible properties of all varieties belonging to, a problem may arise if some of these properties are incompatible or exclusive. Thus, selecting the properties for an object is necessary and must take into account those incompatibilities. To deal with this situation, authors described the alternation of properties or classes. Properties alternation is a concept that deals with variety of properties, whereas classes' alternation is an intermediate form of specialisation between simple specialisation and multiple one.

*Alternation of properties* mainly consists of defining in the same class one or more properties in multiple versions. Alternatives of the same property are exclusive. A given object cannot possess more than one version. Thus a definition of properties that are different for objects of the same class is possible. We distinguish alternation of structures and alternation of behaviours.

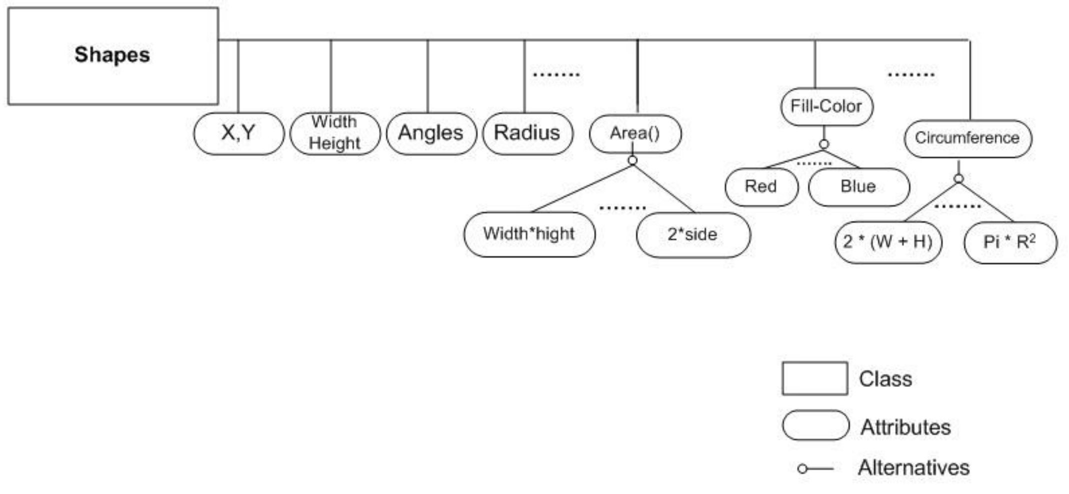


Figure (3-4): Applying Semantic Classification on our Case Study

Figure (3-4) shows how to apply this approach to our case study. With semantic classification, the same hierarchy of classes in figure (2-3) may be reduced to only one class: *Shapes* which holds all possible properties for varieties of objects. Here an alternation appears in the properties *red,..*, and *Blue*. *Fill-Color* is the name of the alternation. Thus the class *Shapes* contains objects that have slightly different structures but the same underlying semantic.

To derive a Rectangle class from the model in Figure (3-4), it may be as the following:

```
Genetic Program Rectangle
{   // has selected the needed attributes:
      X, Y; // to locate rectangle center on X and Y axis.
      Width, Height;
      Angles;
      Area () {Width* Height ;} // has selected one alternative from the area function's alternatives list
      Fill-Color () {Red ;} // has selected one alternative from the Fill-color function’s alternatives list
      Circumference () {2*(Width+ Height)}; // has selected one alternative from the Circumference
                                                         //function’s alternatives list
}
```

And,

```
Shapes Rectangle R
// Where R is developed from Shapes Genome by the Rectangle Genetic Program.
```

The determination of properties that an object of the class holds is based on the interpretation of a program associated with that object. This program is called genetic program. It is composed of a set of rules that reject or select properties from different classes of the Is-A hierarchy resolve conflicts at the same time.

In conclusion, (J.Meslati and S.Ghoul, 1997) work had reduced the number of classes where objects that are semantically equivalent but have different syntax are belonging to same class. It also proposed a resolution to solve names conflict situations by using alternation concept. But this approach did not propose a full bio-inspired model. The semantic classification was done on “is-a” hierarchy model where we aim to work on “Composed by” model. Also, the genetic relations between genes (features) are not modeled.

The closest works to our approaches are (S.Ghoul, 2010) and (S.Ghoul, 2011). Where, in the first work, several general principles to model the genetic selective inheritance were proposed, our work is mainly a development and formalization of some of these general guidelines. This work is a general platform for the genome modeling principles, where the genotype (class variants) is a genome with Enabled/Disabled traits. After that, objects are instantiations of those variants.

In our work we will concern and develop only two of the principles introduced in (S.Ghoul, 2010) work:

- Each artificial entity is a Phenotype developed from a Genotype of a Genome.
- The nature of a Genotype is completely operative.

The work (S.Ghoul, 2011) proposed a bio-inspired approach to support Aspect-Oriented paradigm. The author has used several genetics concepts including Genome, Genotype, and Phenotype to support the Aspect-Oriented design. According to that, the author proposed an extension to OOPL. This extension allows the definition of:

- Different versions of data and methods in the same class.
- Versions compatibility rules, and
- Selective inheritance (definition of configurations).

The following examples (S.Ghoul, 2011) illustrate these extensions.

- **Aspect Class Implementation** (Several versions support method multiple definitions).

Each aspect class interface may be implemented by several aspect class implementations (versions). Each implementation (as seen below) supports method multiple definitions (source code).

```
Set: (Aspect Class= Implementation, Order=first,
state =experimental)
{// shared methods
  Set () = (scope=shared) {// create set source code};
  ~ Set () = (scope=shared) {// destroy set source
code};

//Multi-defined methods: static (st) & Dynamic (Dy)
//behavior (Bh)
Void initialize ():(Bh=st){ rear = 0};
Void initialize ():(Bh=Dy){ rear = null};

Bool Empty (): (Bh=st){return (rear = 0)};
Bool Empty (): (Bh=Dy){return (head= null)};
…
} // End Set: Implementation
```

```
Set: (Aspect Class= Implementation, Order=last,
state =correct)
{// shared methods
  Set () = (scope=shared) {// create a set};
  ~ Set () = (scope=shared) {// destroy a set};

//Multi-defined methods: static (st) & Dynamic (Dy)
//behavior (Bh)
Void initialize ():(Bh=st){ rear = -1};
Void initialize ():(Bh=Dy){ rear = null ; head=null};

Bool Empty (): (Bh=st){return (rear = = -1)};
Bool Empty (): (Bh=Dy){return (head = = null)};

…
} // End Implementation
```

**- Aspect Class Control**

An aspect class control is composed of logical assertions ensuring the coherence of aspects inside an aspect class interface, and in the whole aspect class interfaces hierarchy (the whole design)

```
Set: (Aspect Class= Control)
{
Type
          {
          Aspect Class = <{interface, implementation, control, configuration, global, client}, 1>}
          Scope   = <{shared, sparated},1> ;
          Bh       = <{st,Dy},1> ;  // Defining behavior alternatives.
          Datastr  = <{st,Dy,Temp,pers},2> ;  // Defining Datastr alternatives.
          Order    = <{first, last, experimental },1> ; // Defining Order alternatives.
          State    = <{correct, experimental },1>; // Defining State alternatives.
          }
Exclude // Defining the Exclude relations
          {
          (Bh=Dy) <¬>  (Bh=st);
          (Datastr=st) <¬> ( Datastr= Dy)
          (Datastr=Temp) <¬> ( Datastr= Pers);
          }
Imply // Defining the Imply relations
          {
          (Bh=Dy) → (Datastr = Dy);
          (Bh=st) → (Datastr = st);
          }
Default  // Defining the Default relations
          {
          (Bh=st), (Datastr = st, Temp), (state=correct), (order= last)
          }
}// End Set: Control
```

## - Aspect Configurations

An aspect class configuration includes different configurations of object by composing aspects. Each configuration ensures the generation of objects with the aspect it encompasses.

```
Queue: (Aspect Class= Configurations)
{
          Config: (Name=Default){};

          Config: (Name=CL_Aspect)
                    {
                      Require {(View=CL)};
                      Imply {(DataStr=Persistent)};
                    }// End CL_Aspect

          Config: (Name=LL_Aspect)
                    {
                      Require {(View=LL)};
                      Reject {Size, Full()}
                    }// End LL_Aspect
}// End Queue: Definition
```

So, this work uses the precedent extensions in its implementation. In our work we will develop these extensions

# CHAPTER FOUR

# A GENETICS-BASED APPROACH TO INHERITANCE MODELING

As we mentioned, in our approach to inheritance modeling we aim to be as close as possible to the real world. For that, we inspired from the genetics processes which are the basis of living organisms. This was to solve many problems that object oriented languages did not solve, or to modify some concepts to be more efficient in use.

Selective inheritance, which is one of the inheritance properties, that is not supported by any of object oriented languages, is offset by the genetic genotyping process which is the main process in producing organisms with different characteristics in our real life. Both, selective inheritance and genotyping are the processes of selecting desired traits and functions from a set of all possible characteristics.

In our work, we will develop and formalize in depth the genetic concepts introduced in (S.Ghoul, 2010).Our study is limited mainly to the two following principles:

- Each artificial entity is a Phenotype developed from a Genotype of a Genome.
- The nature of a Genotype is completely operative.

## 4.1 Some New Definitions

In the following Table (4-1), we will present some basic conventional OO concepts and their new redefinitions in our work: Class, attribute, method, composition, inheritance, and instance.

| | Conventional OO Concepts | Genetics-Based Concepts |
|---|---|---|
| **Class** | A class is a set of objects that share common attributes. In other words, the class groups objects that have similar structural and behavioural properties. Objects that have slightly different profiles must belong to different classes. The following definition shows its syntactic structure.<br>**Class** *class-name*<br>**{**<br>// attributes definition<br>// methods definition<br>**}** | We use the semantic class definition introduced in (J.Meslati and S.Ghoul, 1997), where the semantic class groups objects which may have different structural and behavioural properties. But each object owned properties are subset of its class properties. So, a semantic class groups together objects holding subsets of its properties Figure. The following definition shows its syntactic structure.<br>**Class** *class-name*<br>**{**<br>// attributes alternative definition<br>// methods alternative definition<br>**}**<br>In addition, we have added some new concepts as is it follows:<br>**Class** *class-name*<br>**{**<br>// attributes alternative definition<br>// methods alternative definition<br>***Control Rules***<br>*//definition of rules that govern the classification.*<br>*// definition of rules that control the coherent selection of object properties from the alternatives.*<br>**}** |
| **Attribute** | It is a specification that defines a data structure of an object. Values of attributes form an object **state**. Each attribute is defined by a single data structure as it follows:<br>**Class** *class-name*<br>**{**<br>*// Attributes:*<br>Date-Type$_i$ att$_i$;<br>..<br>// *Methods*: } | Each attribute has a set of alternative definitions. An object may hold coherent alternatives from each “needed” attribute. It is defined as it follows:<br>**Class** *class-name*<br>**{**<br>*// Attributes:*<br>*Att$_i$= alternatives {alt$_1$,alt$_2$... alt$_n$};*<br>*//Methods*<br>**}** |

| | | |
|---|---|---|
| **Method** | Subroutine (or *function*) associated with a class. In other words, it is an action which an object is able to perform. It is defined by a single way as the following:<br>**Class** class-name<br>**{**<br>*// Attributes:*<br>// *Methods*:<br>Date-Type$_i$ meth$_i$(){ *// method body*}<br>**}** | Each method (function) has a set of alternative definitions. An object may hold coherent alternatives from each “needed” attribute. It is defined as it follows:<br>**Class** *class-name*<br>**{**<br>*// Attributes:*<br>…<br>*//Methods:*<br>*Meth$_i$()= alternatives {met$_1$,met$_2$... met$_n$};*<br>**}** |
| **Composition** | Composed class uses instance variables that refer to other objects (simple or composed). Each class instances holds these "referred to" objects. The following example illustrate a “has a” relationship in pseudo code:<br>**Class** brick { .. }<br>**class** wall<br>**{**<br>// Attributes:<br>Brick brick1, brick2;<br>// Methods:<br>wall() // Constructor<br>{<br>this.brick1 = new brick();<br>this.brick2 = new brick();<br>}<br>**}**<br>As we can see, “wall” contains a number of brick attributes. We want each of these attributes to be a “brick” object. To do this, we simply instantiate them within the constructor of the “wall” class. Each of these brick classes will function as a normal class but also as an attribute of “wall.” | A semantic class definition may be composed of other semantic classes’ definitions. An instance of this class may hold only selected attributes and methods from selected "composed by classes". So, It may not include a "composed by" class components at all, include some part of it, or include it completely (with one selected alternative for each component).<br>The following example illustrates this definition.<br>**class transportation**<br>**{**<br>**class aerial** { };<br>**class maritime** { };<br>**class ground** { }<br>…<br>**}**<br>The class Transportation definition is composed by the classes: aerial, maritime, and ground definition; which is different from the attribute composition concept in the conventional object oriented paradigm.<br>Each instance may hold its properties selected from any one of these classes, i.e. the object T$_i$ defined as it follows: |

| | | |
|---|---|---|
| | | **Transportation** selection$_i$ $T_i$<br>Holds properties defined by the selection Selectioni |
| **Inheritance** | It is based on the “Is-a” relation where a subclass inherits all the attributes and methods of its parent class(s) Figure (4-1/a). | It is based on a selective inheritance program that selects only needed attributes/methods from a specific class and its "composed by" classes. This selection program replace the "Is a" relation, which define implicitly a total inheritance, whereas this program defines an explicit and restrictive inheritance, Figure (4-1/b). |
| **Instance** | An object is a value of a class, called an instance of the class and has the behaviours of its class. According to that, all objects instantiated from a class are similar; where they all will have the same structure (attributes /methods). | An object (O) may be created from a semantic class (C), according to an explicit program (P) that selects its needed attributes and methods alternatives from (C). According to that, objects created from the same class may differ from each other. Example:<br>**Transportation** selection$_1$ $T_1$ // aerial<br>**Transportation** selection$_2$ $T_2$ // maritime & aerial. |

Table (4-1): Comparison between Some Conventional OO and Genetics-Based Concepts.

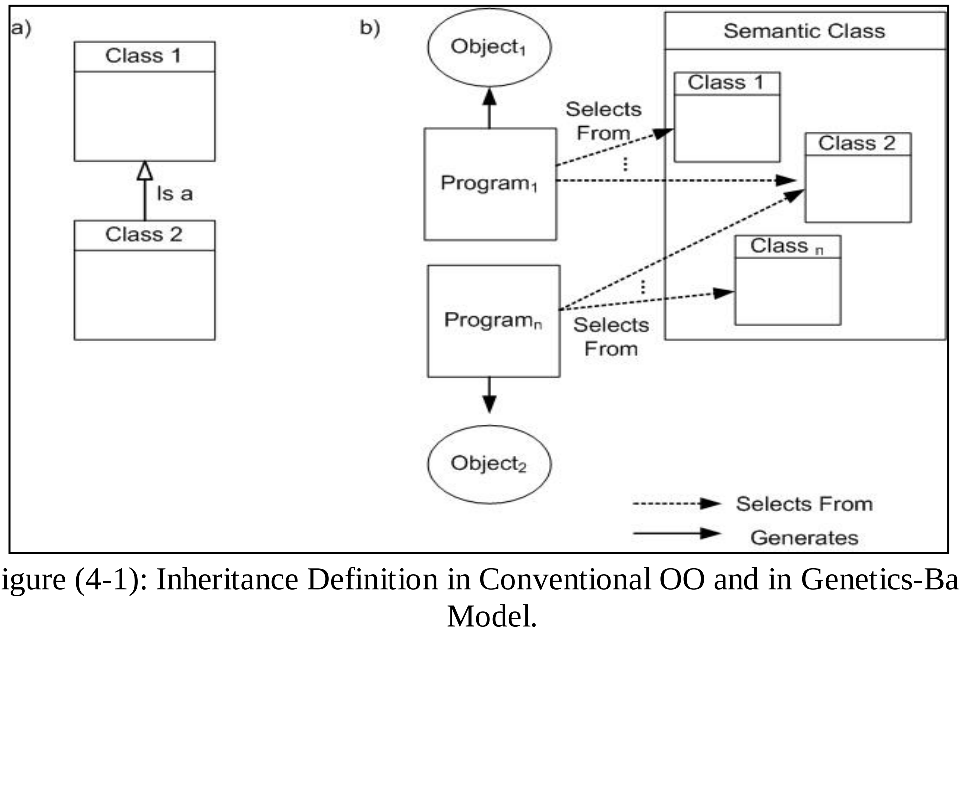


Figure (4-1): Inheritance Definition in Conventional OO and in Genetics-Based Model.

## 4.2 Each Artificial Entity is a Phenotype Developed from a Genotype of a Genome

In the following we will develop a Genome and a Genotyping models from the selective inheritance point of view.

### Genome

The genetic patrimony, genome, of a species includes the definition of all its possible characteristics (organic, functional, and behavioral) along with the information controlling their coherence (S.Ghoul, 2010).

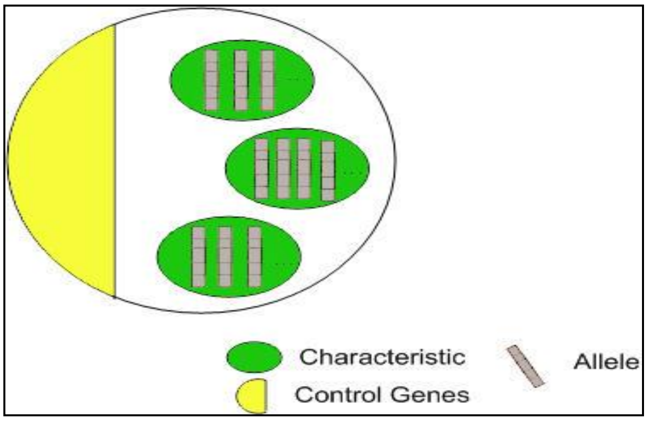


Figure (4-2): Genome Model, Versions of Characteristics.

Each characteristic might be developed in alternative ways. Each way constitutes a version of this characteristic Figure (4-2). So, a characteristic is defined by an allele of genes; each one is responsible for the development of a version. The physical development and phenotype of organisms can be thought of as a product of genes interacting with each other and with the environment.

We detail the above general Genome model Figure (4-2), by the following "composed by" inheritance Modeled specification.

Genome= {Species$_i$, Species -Control-Genes}, i=1 to n.
*// Genome "Semantic Class" is Composed of a set of Species "Semantic Classes" controlled by Species Control Genes.*

Species$_i$ = {Specie-Architecture$_j$}, j=1 to n.
*// Specific Species is composed of a set of Specie Architectures.*

Species-Control-Genes$_j$ = {AreDominantSpecie}.
*// To specify the dominant Specie inside the Genome.*

Specie-Architecture$_j$= {Specie-Architecture-Genes$_k$, A*rchitecture-Control-Genes*}, k=1 to n.
*// Each Specie-Architecture composed of a set of Specie-Architecture-Genes that are controlled by Architecture Control Genes.*

Specie-Architecture-Genes$_k$= {Organ$_y$, Organ-Control-Genes}, y=1 to n.
*// Each Specie-Architecture-Genes is composed of a set of Organs that are controlled by Organ Control Genes.*

A*rchitecture-Control-Genes*= {AreImpliedInArchitecture, AreDefaultInArchitecture, AreExcludedFromArchitecture}.
*// The set of genes that ensure the compatibility inside a Specie-Architecture.*

Organ$_y$= {Phylogenies-Genes, Ontogenesis-Genes, Epigenesis-Genes, POE-Control-Genes}.
*// Each Organ "Attribute" is composed of a set of POE Genes that are controlled by POE- Control Genes.*

Organ-Control-Genes= {PerformSameFunctionOrgans, AreImpliedOrgans, AreExcludedOrgans}
//*The set of Control genes that ensure the compatibility between Organs.*

Ontogenesis-Genes = {Organ-Definition-Genes, Organ-Functions-Genes, Organ-Behavior-Genes, Ontogenesis-Control-Genes}.
*// The set of genes that specify the definition, function and behavior of an organ.*

POE-Control-Genes= {AreRelatedToEvolution, AreRelatedToConstruction,
AreRelatedToLearning}.
*// The set of control genes that classify the genes according to POE axis.*

Organ-Definition-Genes= {Coding-Genes, None-Coding-Genes}.
*// Definition genes is composed of two sets: Coding and None coding. We concern about the Coding genes.*

Coding-Genes= {D-Gene$_m$}, m=1 to n.
*// The set of genes that construct the Organ.*

Organ-Functions-Genes= {F-Gene$_q$}, q = 1 to n.
*// The set of genes that construct the Organ Functions.*

Organ-Behavior-Genes= {B-Gene$_r$}, r = 1 to n.
*// The set of genes that construct the Organ Behavior*

Ontogenesis-Control-Genes= {AreRelatedToAspect, AreExclusive, AreImplied, AreDominant, AreGenotype}.

*//The set of Control genes that ensure the compatibility between definitions, functions and behaviors of an Organ.*

D-Gene$_m$= {D-Alternatives}.

*//Each Organ Definition Gene has a set of alternatives to be chosen.*

F-Gene$_q$= {F-Major-Alternatives, F-Sub-Alternatives}.

*//Each Organ Function Gene has two sets of alternatives: Major set to choose the main alternative from it and a Sub set to choose from it if needed.*

B-Gene$_r$= {B-Major-Alternatives, B-Sub-Alternatives}.

*//Each Organ Behavior Gene has two sets of alternatives: Major set to choose the main alternative from it and a Sub set to choose from it if needed.*

AreRelatedToAspect= {(D-gene, F-gene, B-gene); O$_1$: Organ | $\forall$ D-gene, F-gene, B-gene $\in$ O$_1$}

*// AreRelatedToAspect genes: link together the characteristics that are related to a same aspect. For example, the characteristics related to male sex aspect, the characteristics related to female sex aspect, etc.*

Are Exclusive= {gene$_1$: AreRelatedToAspect; gene$_2$, gene$_3$: Specie-Architecture-Genes$_1$; alt1, alt$_2$: ALTERNATIVES | alt$_1$ <> alt$_2$ if alt$_1$, alt$_2$ $\in$ gene1 $\vee$ gene$_2$ <> gene$_3$ if gene$_3$ *contradict* gene$_2$}, where <> means: exclude, ALTERNATIVES: is any of the gene alternatives sets.

*// These genes identify the characteristics that are exclusive. A characteristic excludes another if they are alternatives (elements of the same alternation) or they are incompatible. For example, the color blue excludes the color green when they concern eyes (elements of the same alternation) and the beard excludes the female sex (incompatible).*

AreImplied= {gene$_1$, gene$_2$: AreRelatedToAspect; ph: PHENOTYPE | gene1 $\rightarrow$ gene$_2$ if gene$_1$ $\in$ ph $\rightarrow$ gene$_2$ $\in$ ph}, where PHENOTYPE is the physical instance of the Genotype, and $\rightarrow$ means: Imply.

*// A characteristic implies another if its presence in a phenotype implies the presence of the other. For example, human male sex implies beard and the gruff voice etc.*

AreDominant= { $f_1$ :N ⤀ ALTERNATIVES $\vee$
$f_2$: X ⤀ Coding-Genes,
$f_3$: Y ⤀ Organ-Functions-Genes,
$f_4$: Z ⤀ Organ-Behavior-Genes
| alt$_n$ $\in$ ALTERNATIVES; gene$_x$ $\in$ Coding-Genes;
gene$_y$ $\in$ Organ-Functions-Genes; gene$_y$ $\in$ Organ- Behavior-Genes
Dom $f_1$=1..n, Dom $f_2$=1..X, Dom $f_3$=1..Y, Dom $f_4$=1.. Z }

*// AreDominant. This relation exists between exclusive characteristics. Dominate orders them according to their importance and the most dominant must be selected for a phenotype. If, for any reason, this is not possible, the next (in dominance) is chosen. For example, the dominance relation between the colors of eyes may be specified as {black, brown, blue, etc.}.*

AreGenotype= {$gene_1$… $gene_{\mathbf{n}}$**:** AreRelatedToAspect; g: GENOTYPE | gene1 $\wedge$ … $\wedge$ $gene_{\mathbf{n}}$ $\rightarrow$ g}, where GENOTYPE is the set of enabled coherent genes.
*// This relation links together genome characteristics that define a genotype.*

Where:

Gene = {Gene-Non-Coding-Information, Gene-Coding-Information}
*// Each Gene Consists of two information sets.*

Gene-Non-Coding-Information = {priority, selection state, activation state}.

*// the set of information that specify the gene state.*

Priority = pri, pri$\in$ Integer
*// Priority identifies the gene order in execution or selection, where dominant gens have the first priority.*

Selection state = (E) enabled | (D) disable|($\perp$) not applicable.
*// To specify the gene state during selection process.*

Activation state = (A) active | (S) silent.
*// To specify whether to Activate the Gene or to keep it Silent.*

Gene-Coding-Information = {organic instructions, functional instructions, control instructions}.
*// specifying the instructions that determine whether the Gene is for control or construction.*

To simplify this model, Figure (4-3) introduces a tree which explains the genome structure.

From this model Figure (4-3) and the formalization, we suppose that the **Genome** is a Class that consists of all existing *species* (Humans, Animals, Plants, Microscopic… etc), beside the *species-Control-genes* to ensure that one species will be selected. Each species has its own *architecture* (construction genes, control genes…) and *architecture-Control-Genes* that classify each architecture with its correct related genes. Each *Specie-architecture-Gene* contains a list of all *organs* that are related to that Specie-architecture and *organ-control-genes* that control this organs classification. In each organ, there are lists of its *Phylogenies*, *Ontogenesis* and *Epigenesis genes* beside the *POE-Control-Genes* that ensure the correct classification so that Phylogenies-genes will contain genes responsible about evolution, Ontogenesis-genes will contain genes responsible about construction and finally,

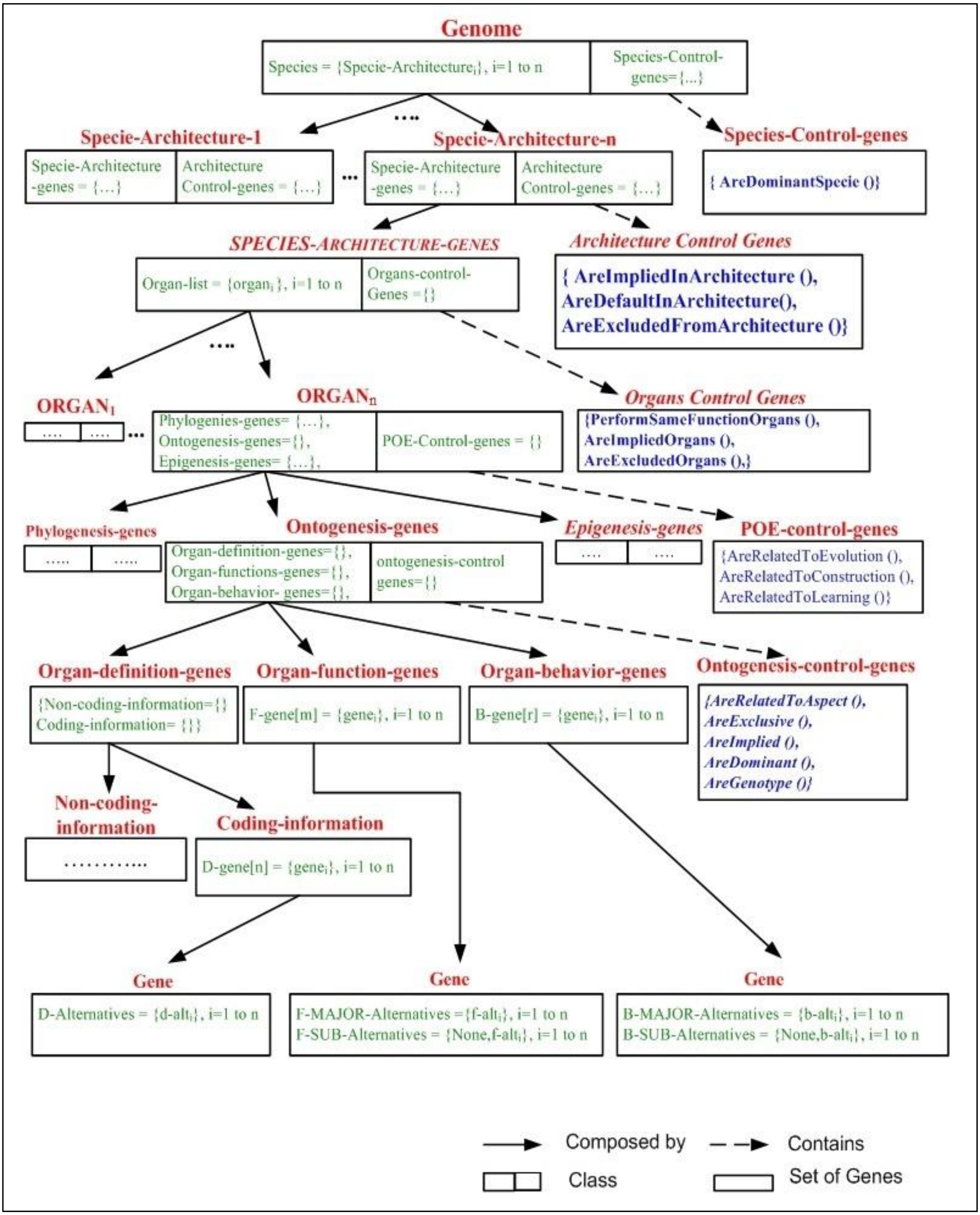

Genome
Species = {Specie-Architecture$_i$}, i=1 to n
Species-Control-genes={...}
....
Specie-Architecture-1
Specie-Architecture -genes = {…}
Architecture Control-genes = {…}
...
Specie-Architecture-n
Specie-Architecture -genes = {…}
Architecture Control-genes = {…}
Species-Control-genes
{ AreDominantSpecie ()}
SPECIES-ARCHITECTURE-GENES
Organ-list = {organ$_i$ }, i=1 to n
Organs-control-Genes = {}
Architecture Control Genes
{ AreImpliedInArchitecture (),
AreDefaultInArchitecture(),
AreExcludedFromArchitecture ()}
....
ORGAN$_1$
.... ....
...
ORGAN$_n$
Phylogenies-genes= {…},
Ontogenesis-genes={},
Epigenesis-genes= {…},
POE-Control-genes = {}
Organs Control Genes
{PerformSameFunctionOrgans (),
AreImpliedOrgans (),
AreExcludedOrgans (),}
Phylogenesis-genes
..... .....
Ontogenesis-genes
Organ-definition-genes={},
Organ-functions-genes={},
Organ-behavior- genes={},
ontogenesis-control genes={}
Epigenesis-genes
.... ....
POE-control-genes
{AreRelatedToEvolution (),
AreRelatedToConstruction (),
AreRelatedToLearning ()}
Organ-definition-genes
{Non-coding-information={}
Coding-information= {}}
Organ-function-genes
F-gene[m] = {gene$_i$}, i=1 to n
Organ-behavior-genes
B-gene[r] = {gene$_i$}, i=1 to n
Ontogenesis-control-genes
{AreRelatedToAspect (),
AreExclusive (),
AreImplied (),
AreDominant (),
AreGenotype ()}
Non-coding-information
…………....
Coding-information
D-gene[n] = {gene$_i$}, i=1 to n
Gene
D-Alternatives = {d-alt$_i$}, i=1 to n
Gene
F-MAJOR-Alternatives ={f-alt$_i$}, i=1 to n
F-SUB-Alternatives = {None,f-alt$_i$}, i=1 to n
Gene
B-MAJOR-Alternatives = {b-alt$_i$}, i=1 to n
B-SUB-Alternatives = {None,b-alt$_i$}, i=1 to n
Composed by
Contains
Class
Set of Genes

Figure (4-3): Genome Structure.

Epigenesis-genes list will contain genes responsible about learning and immune. The part that concerns us is the *Ontogenesis-genes.* It contains all genes that construct the organ which are: *Organ-Definition-Genes, Organ-Function-Genes* and *Organ-Behavior-Genes.* Each *gene* consist*s of alternatives* to select one from them, but the overall chosen alternatives must be coherent and compatible.

## Genotype

Genotype is the genetic makeup, as distinguished from the physical appearance (phenotypes), of an organism or a group of organisms. This makeup is a combination of alleles of genes that determines a specific characteristics or traits. The chosen characteristics must be coherent. So, to ensure and control this coherence, the *controls genes* in the genome establish and manage dependencies relations between characteristics.

In the following we will introduce the dependencies relations for the most common identified control genes:

o *AreExclusivegenes: These genes* ensure the exclusion between Enabled and Disabled genes. The genotype, being coherent by construction, holds normally non exclusive organs and functions. However, some specific evolution states may necessitate exclusion between organs and between functions. This is defined by the following rules:

- *Enable organ/definition/function/behavior genes <>*
  *Enable organ/definition/ function/behavior genes*
- *Disable organ/definition/function/ behavior genes <>*
  *Disable organ/ definition/function/ behavior genes*

Enabling/Disabling an organ/definition gene may exclude enabling/disabling other organs/definition genes which are, for the evolution semantic, needed to be excluded. Enabling/Disabling a function/behavior gene may exclude enabling/disabling other functions/ behavior which are for the evolution semantic, needed to be exclusive (replacement, incompatibility . . .).

○ *AreImpliedgenes.* These genes ensure the implication between Enabled and Disabled genes. This is supported by the following rules:

- *Enable organ/definition/function/behavior genes* → *Enable organ/definition/ function/behavior genes*
- *Disable organ/definition/function/ behavior genes* → *Disable organ/ definition/function/ behavior genes*

Enabling/Disabling an organ/definition gene may imply enabling/disabling others organs/definition genes (which are related) and their functions genes.

Enabling/Disabling a function/behavior gene may imply enabling/disabling others functions/behavior genes (with which it collaborates) and their organs definition genes.

## Phenotype

A ***phenotype*** is an instance of a given genotype Figure (4-4). In the artificial world, several phenotypes may be instances of the same genotype. The source code of Windows Vista and that of Windows XP are two genotypes of Windows Species. All the executable codes produced from the source code of Window Vista are phenotypes of this genotype, and all the executable codes produced from the source code of Window XP are phenotypes of this genotype.

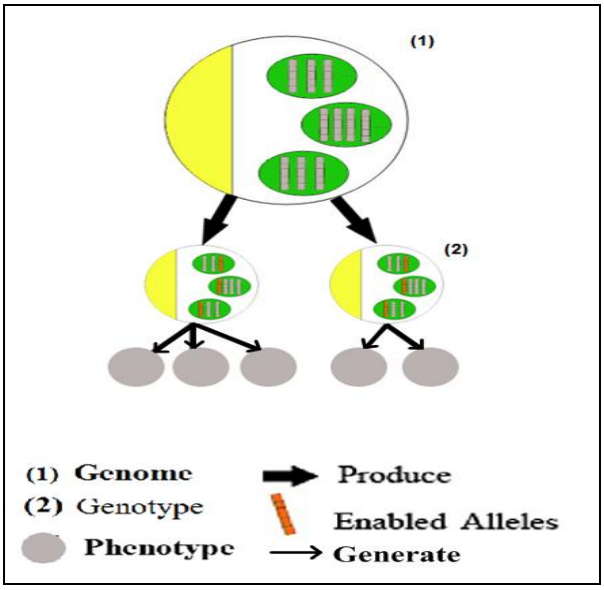


Figure (4-4): Genome, Genotype and Phenotype

## 4.3 The Nature of a Genotype is completely operative

In this section we present the genes' relations model of the Genotype as shown in Figure (4-5), see (S. Ghoul, 2010).

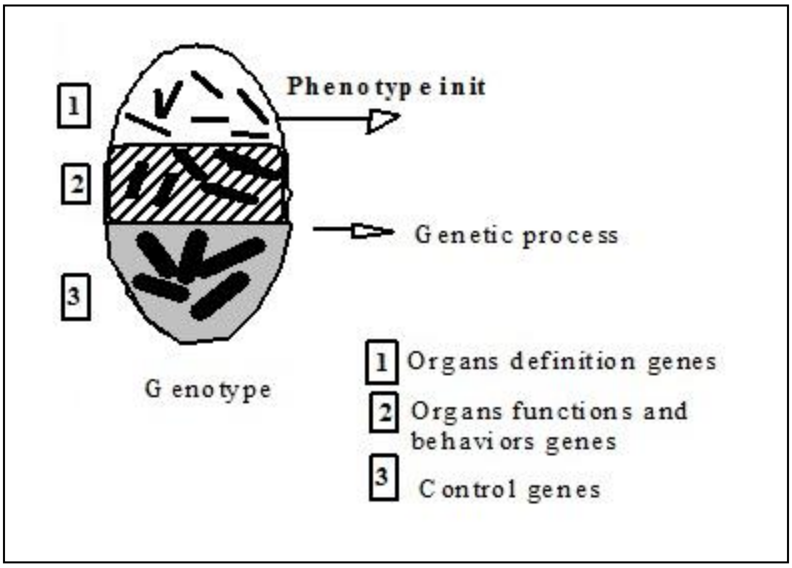

1
2
3
Phenotype init
Genetic process
Genotype
1 Organs definition genes
2 Organs functions and behaviors genes
3 Control genes

Figure (4-5): Genotype

The genotype Figure (4-5) consists of:

- *Organic genes* offering only one way (allele) for developing each organ, and generating a phenotype,
- *Functional genes* offering only one way for each organ function and behavior, and
- *Control operators*, as those of a species but without the genes controlling the species evolution and genotype definition. All the genotype genes are silent, except those controlling the phenotype initialization (Phenotype Init process).

In our work, we explicitly defined a genotype as it follows:

```
Genotype Def genotype-name
{
  Specie-Name;
        // we specify a Specie.
  Specie-Architecture;
        // we specify the Architecture.
  Enable <Organs definition genes, functions definition genes>
        // we specify what to be enabled.
  Disable <Organs definition genes, functions definition genes>
        // we specify what to be disabled.
}
```

We must specify the name of the wanted Specie, the wanted Architecture and we select required organs characteristics and reject undesirable organ characteristics from a genome, Where:

*Enable:* Allows a phenotype of this genotype to hold a set of explicitly enumerated organs and functions definition genes. This set is implicitly augmented by the genetically implied genes.

*Disable:* Allows a phenotype to lose a set of explicitly enumerated organs and functions definition genes. This set is implicitly augmented by the genetically implied genes. The disabled properties are inactive for that phenotype.

The interpretation of a genotype definition is mainly supported by the control genes which ensure the coherence of the definition process.

The interpretation of the genotype definition enforces the following rules:

- *Initial state: Elements and coherence*

R1. The *Genome* holds the set of all Specie-Architectures. Each Specie-Architecture holds an initial set of all its organs with their definitions, functions, and behaviors genes generated genetically at Specie selection.

R2. Let EnabOrg, EnabDef, EnabFun and EnabBeh be, respectively, the lists of the organs, organs-definitions genes, functions genes, and behaviors genes to be *enabled*.

*EnabOrg ← Organs (imposed in Enable clause);*
*EnabDef ← Organs-definitions (imposed in Enable clause);*
*EnabFun ← Functions (imposed in Enable clause);*
*EnabBeh ← behaviors (imposed in Enable clause);*

The coherence of EnabOrg, EnabDef, EnabFun, and EnabBeh is checked separately because there is no *Exclude* relation between *Enabling* Organs, *Enabling* Organs-definitions, *enabling* functions and *enabling* behaviors. This coherence deals especially with:

**(1)** The existence of the Organs, Organs-definitions, functions and behaviors in the associated Specie-Architecture, and,

**(2)** The verification that the elements of each list do not exclude elements of the same list.

R3. Let DisabOrg, DisabDef, DisabFun and DisabBeh be, respectively, the lists of Organs, Organs-definitions, functions and behaviors genes to be *Disabled*.

*DisabOrg ← Organs (discarded in Disable clause);*
*DisabDef ← Organs-definitions (discarded in Disable clause);*
*DisabFun ← Functions (discarded in Disable clause);*
*DisabBeh ← behaviors (discarded in Disable clause);*

The coherence of DisabOrg, DisabDef, DisabFun and DisabBeh is checked separately because there is no *Exclude* relation between *Disabling Organs*. *Disabling* Organs-definitions, *disabling* functions and *disabling* behaviors. This coherence deals especially with:

**(1)** The existence of the Organs, Organs-definitions, functions and behaviors in the associated Specie-Architecture.

**(2)** The verification that the elements of each list do not exclude elements of the same list.

R4. The coherence of EnabOrg with DisabOrg is checked: EnabOrg∩DisabOrg = Ø, each element of EnabOrg doesn't *imply* directly or indirectly an element of DisabOrg, and each element of DisabOrg doesn't imply directly or indirectly an element of EnabOrg

The coherence of EnabDef with DisabDef is checked:EnabDef∩DisabDef =Ø, each element of EnabDef doesn't *imply* directly or indirectly an element of DisabDef, and each element of DisabDef doesn't imply directly or indirectly an element of EnabDef.

The coherence of EnabFun with DisabFun is checked: EnabFun∩DisabFun = Ø, each element of EnabFun doesn't imply directly or indirectly an element of DisabFun, and each element of DisabFun doesn't imply directly or indirectly an element of EnabFun.

The coherence of EnabBeh with DisabBeh is checked: EnabBeh ∩ DisabBeh= Ø, each element of EnabBeh doesn't imply directly or indirectly an element of DisabBeh, and each element of DisabBeh doesn't imply directly or indirectly an element of EnabBeh.

- *Enable list processing by scanning genetic relations*

R5. The processing of Enable list is obtained, by scanning the Imply relation according to the dominate order, as it follows:

1. For each element in the EnabOrg, not yet processed, find the Enabled Organs and put them in EnabOrg.
2. For each element in the EnabOrg, not yet processed, find the Excluded Organs and put them in DisabOrg.
3. For each element in the EnabDef, not yet processed, find the Enabled (1) definitions and put them in EnabDef, (2) functions and put them in EnabFun and (3) behaviors and put them in EnabBeh.
4. For each element in the EnabDef, not yet processed, find the Excluded definitions and put them in DisabDef.
5. For each element in the EnabFun, not yet processed, find the Enabled (1) definitions and put them in EnabDef, (2) functions and put them in EnabFun and (3) behaviors and put them in EnabBeh.
6. For each element in the EnabFun, not yet processed, find the Excluded functions and put them in DisabFun.
7. For each element in the EnabBeh, not yet processed, find the Enabled (1) definitions and put them in EnabDef, (2) functions and put them in EnabFun and (3) behaviors and put them in EnabBeh.
8. For each element in the EnabBeh, not yet processed, find the Excluded behaviors and put them in DisabBeh.

We remind that these imply relations have the following form:

*Enable organ/organ-definition /function/behavior genes → Enable organ /organ-definition/function/ behavior genes*

*Enable organ/organ-definition /function/behavior genes <> Enable organ /organ-definition/function/ behavior genes*

- *Disable List processing by scanning genetic relations*

R6. The processing of Disable lists is obtained, by scanning the genetic relations according to the dominate order, as it follows:

1. For each element in the DisabOrg, not yet processed, find the Excluded organs and remove them from EnabOrg.

**2.** For each element in the DisabOrg, not yet processed, find the Disabled organs and put them in DisabOrg.

**3.** For each element in the DisabDef, not yet processed, find the Excluded definitions and remove them from EnabDef.

**4.** For each element in the DisabDef, not yet processed, find the Disabled (1) definitions and put them in DisabDef, (2) functions and put them in DisabFun and (3) behaviors and put them in DisabBeh.

**5.** For each element in the DisabFun, not yet processed, find the Excluded functions and remove them from EnabFun.

**6.** For each element in the DisabFun, not yet processed, find the Disabled (1) definitions and put them in DisabDef, (2) functions and put them in DisabFun and (3) behaviors and put them in DisabBeh.

**7.** For each element in the DisabBeh, not yet processed, find the Excluded behaviors and remove them from EnabBeh.

**8.** For each element in the DisabBeh, not yet processed, find the Disabled (1) definition sand put them in DisabDef, (2) functions and put them in DisabFun and (3) behaviors and put them in DisabBeh.

We remind that these imply relations have the following form:

*Disable organ/organ-definition /function/behavior genes → Disable organ /organ-definition/function/behavior genes*

*Disable organ/organ-definition /function/behavior genes <> Disable organ /organ-definition/function/ behavior genes*

- *Loop on Enable list and Disable list processing*

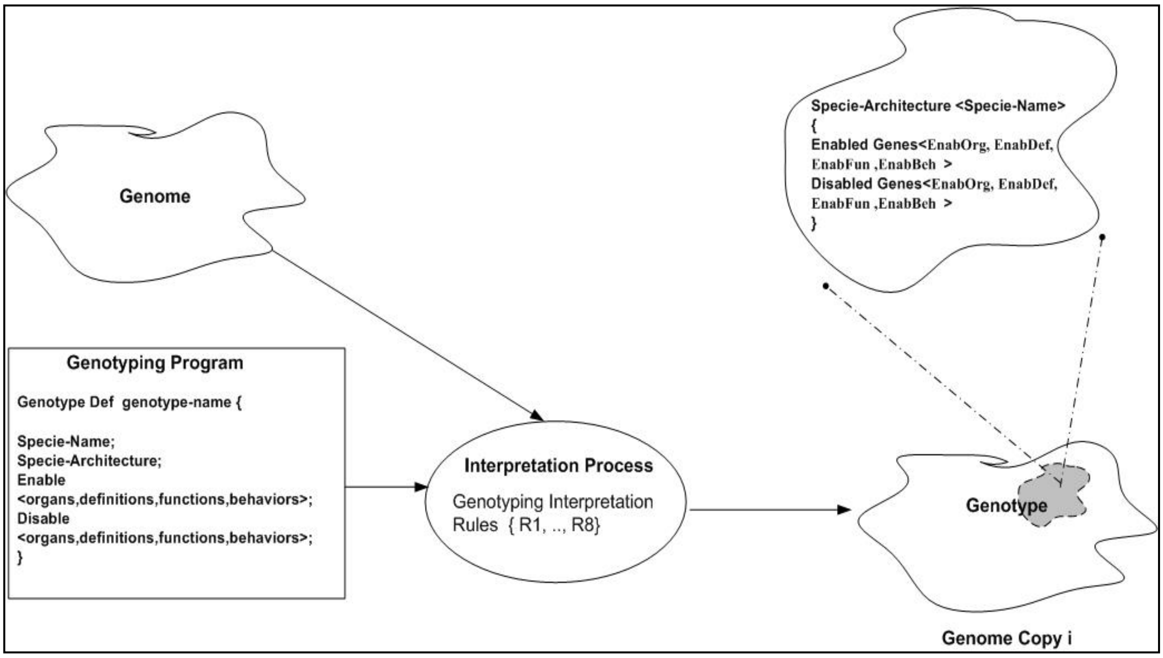
R7. Repeat R5 and R6 until all their elements are processed.

- *Final state*

R8. The result of this interpretation may be one of the following:

- A Failure if coherence errors were found.
- A genome copy if the interpretation successes. This copy contains the obtained genotype defined by:
  - The Enabled genes of Organs, Organs-Definitions, Functions and Behaviors.
  - The Disabled Organs, Organs- Definitions, Functions and Behaviors.

This Interpretation process of the Genotype Program is shown in Figure (4-6).



```
Genotyping Program

Genotype Def  genotype-name {

Specie-Name;
Specie-Architecture;
Enable
<organs,definitions,functions,behaviors>;
Disable
<organs,definitions,functions,behaviors>;
}
```

```
Specie-Architecture <Specie-Name>
{
Enabled Genes<EnabOrg, EnabDef,
EnabFun ,EnabBeh  >
Disabled Genes<EnabOrg, EnabDef,
EnabFun ,EnabBeh  >
}
```

Figure (4-6): Interpretation process of the Genotype Program.

When specific *specie* is needed, its name is sent from a *Genotyping-Program*, beside the *genome* class through an *Interpretation Process* to select the related *specie-architecture*. To solve the problem of not selecting any specie, and to ensure that only one specific *specie-architecture* was selected, the parameter is tested with the *species-Control-genes* which provide the *Dominant* Specie, which is a pre-defined value. So, when two parameter values of two different species have been sent to the object Genome, the dominant pre-defined specie is selected, and when a null parameter value has been sent to the object Genome, the dominant pre-defined specie is selected as default specie. This step prevents the rapid suspension of genotyping process, and helps in continuing the process to produce a final result that may satisfy the user.

In *Genotyping-Program,* the *architecture, organs, definitions, functions* and *behavior* must also be specified. And after the *interpretation* which is governed by the rules mentioned before, a genotype is created with wanted and coherent characteristics.

The following algorithm is the translation of the rules above.

At first, a Genome definition introduced as the following: Genome Class:

```
Class Genome
{
        Structure
            Species= {…
                Species-architecture= {…
                        Organs= {…}
                                        }
                        }
         Control Rules
            Specie-control-genes ();

            Architecture-Control-Genes ();

            Organ-Control-Genes ();

            POE-Control-Genes ();

            Ontogenesis-Control-Genes (){…};
}
```

Second, a set of wanted attributes is defined to be enabled, or a set of unwanted attributes is defined to be disabled. This is achieved by: Genotype Program:

```
Genotype genotype-name (output specie-name, specie-arch, enabled,
disabled)

{
        Get specie-name;
        Get specie-arch;
        Enable < Organs || organ-definition || functions || behavior>;
        Disable< Organs || organ-definition || functions || behavior>;
}
```

Then, both the *Genome class* and the *Genotype Program* are set as inputs for the interpretation process to produce the desired *Genotype.*

```
Process  Interpretation-process (input Genome, specie-name, specie-
arch, enabled, disabled)
{
    If specie-name is set to null || more than specie was selected then
        Test-In- Specie-Control-Genes ();
    if organ <> null then
        if to-enable then add organ to Enaborg
        if to-disable then add organ to Disaborg
    if organ-definition <> null then
        if to-enable then add definition to Enabdef
        if to-disable then add definition to Disabdef
    if function <> null then
        if to-enable then add function to Enabfun
        if to-disable then add function to Disabfun
    if behavior <> null then
        if to-enable then add behavior to Enabbeh
        if to-disable then add behavior to Disabbeh
// if organ & organ-definition &function & behavior = Null
// then set the dominant ();

   Visit- species-Architecture (specie-arch; species- architecture );
// will select the architecture for required Specie

  Test-In-Architecture-Control-Genes (specie-arch, species-Architecture;
arc-genes);
// Will output the architecture genes for required Specie.

  Select-coherent-organs (arc-genes, EnabOrg, Disaborg);
// will select and output the compatible organs with enabled ones.

  Select-ontogenesis (Genome, EnabOrg, Enabdef, Enabfun, Enabbeh,
Disaborg, Disabdef, Disabfun, Disabbeh; genotype);
// will select compatible definition, functions and behaviors alternatives
with enabled ones, and produce acopy of Genome with enabled
properties which form the genotype.
 }
```

To see a full possible algorithm, read (Appendix).

It is worth to mention that the previous interpretation process will be applied on the *Genome* which contains the full set of attributes that are classified into several classes according to “composed by” relations and there will not be any inheritance relations between the inner classes. This classification model provides an easy, flexible and more logical selection of properties. Instead of inheriting from several super-classes and several ancestors according to the “is-a” hierarchy, in our approach, an object will select its needed properties from a class that may be an inner class.

From the algorithm, we notice that the selection process is acting in breadth; while searching for the appropriate and compatible category class, and in depth while deepen inside specific category to search for wanted and compatible traits.

## 4.4 Applying into our Case Study

The case study in chapter two, Figure (2-3), shows a conventional inheritance case, as explained before.

By applying our approach, the hierarchy will be replaced by the model shown in Figure (4-7) through the following steps:

- We tend to have a "Genome" class *Geometric-Shapes* that can produce all shapes exist in geometry including *circles, squares, triangles, rectangles* and others.
- Any shape in geometry may be consists of sides or it may consists of curves or it may be a three-dimensional. *So,* our *Geometric-Shapes* will be composed by the following *“Species”: Curved, Polygons* and *Solids* classes, beside the *Species-Control-Genes* that defines the Dominant Species, *for instance*, the dominant = *Curved.*
- As each *“Species” is* composed by several *“Architectures”,* the *Curved* class in turn can be composed by the following “*Architectures”: Regular* shapes (circle, ellipse, etc.) and *Irregular* shapes (Crescent, etc.), *Polygons* also can be composed by several “*Architectures*” classes which are *Quadrant* that contains all shapes that have four sides (square, rectangle, etc.), *More-Quadrant* that contains all shapes that consist of more than four sides (quintuple, hexagonal, etc.) and *Less- Quadrant* that contains shapes that have less than four sides (triangle, etc.), and *Solids* class is composed by *Surface* class that contains all three-dimensional shapes that consists of surfaces (cube, Pyramid, etc.) and *Non-Surface* class that contains all three-dimensional shapes that don’t consists of surfaces (cone, dome, cylinder, etc.).

- Each *Architecture* is composed by several *Species-Architecture-Genes,* for instance, *Regular Architecture* will be classified into *circle* and *ellipse, Irregular Architecture* will be classified into crescent, cycloid, etc. Beside that there are the *Architectures-Control-Genes* that specify the shapes according to their specific architecture.
- Each *Species-Architecture-Genes* is composed by a number of "*Organs*" beside the *Organs-Control-Genes* that group all the relative organs. For instance, the square will be composed by sides, angles, center, etc. the circle will be composed by radius, center, etc. and so on.
- Each Organ contains its *Organ-definition-genes, Organ-function-genes, and Organ-behavior-genes* along with their "Genes" *alternatives.* For instance, the square has four sides, each side has a color that may take one of alternatives' list {blue, black, etc.}, a style {liner, dashed, etc.}, etc. also, square has an area() that may take one of alternatives' list {(height *width), side$^2$ , etc}, and *Ontogenesis-Control-Genes* that links the organs with the following relations: AreExclusiveGenes, AreImpliedGenes, etc.

Producing desired shape can be done by just specifying Species, Specie-architecture along with the required shape characteristics; for example if we want to instantiate a *Rectangle* object, we must specify:

- Specie-Name: *Polygons.*
- Specie-Architecture: *Quadrant.*

Then we need to enable needed Organs "Parts": *sides* with needed characteristics *length ≠ width, color, style…etc.*

After that an *interpretation process* will produce a complete *Rectangle* with the defined properties with all other *compatible* properties.

As we notice, several different rectangles can be instantiated according to the Genotyping program.

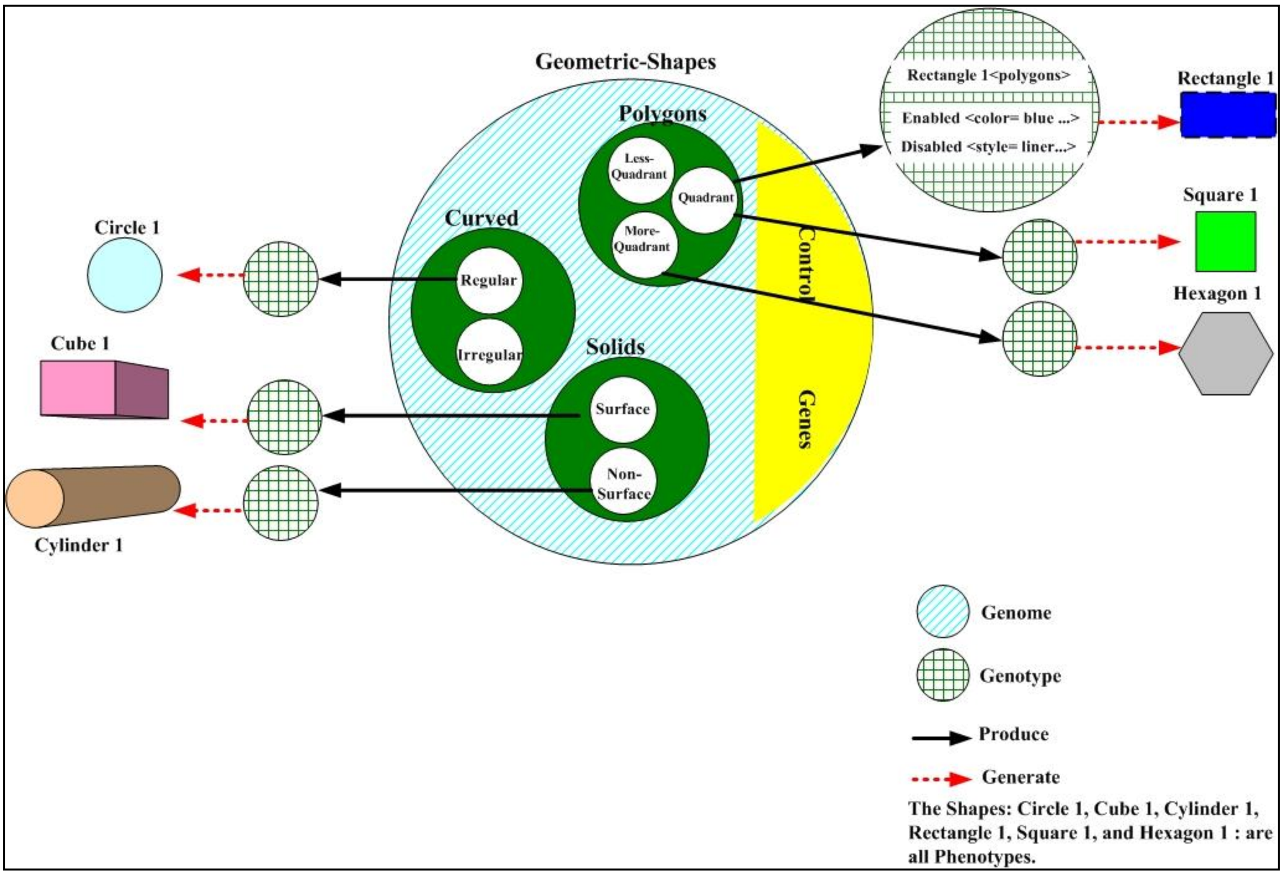


Figure (4-7): Shapes Inheriting Using our "Composed by" Approach.

To implement this, we will use an Object Oriented Languages with some *extension*.

By applying our algorithm, we gain the following:

- Shapes Class modeling:

```
Class Geometric-Shapes
{
  Dominant Curved; // Defines the dominant Species;
  Structure: // to define each Species;
       Curved = { // Defining Species 1 };
       Polygons =     // Defining Species 2;
          {
             Quadrant= // Defining Species2- Architecture1
                { // Defining organs and functions
                  Sides, height…
                  Color= alt {red, blue…} end alt;
                  Line-style= alt {none dashed…};
                  Area () =alt {area1, area2…}
                };
      // defining all Species-Architectures for Species 2
          };
         Solids= { // Defining Species 3 };
  Control-Rules: //Defining the rules that controls the relations
      Specie-control-genes ();
      Architecture-Control-Genes ();
      Organ-Control-Genes ();
      POE-Control-Genes ();
      Ontogenesis-Control-Genes ()
          {
            Enable height → Enable width
            Disable curved → Disable radius
            Enable area (h*w) <> Enable area (2*r)
             ….
            }
}
```

- *Defining requests:*

```
Genotyping Rectangle1 ( )
{
        Polygons; // specifying wanted Species.
        Quadrant; // specifying wanted Species-Architecture.
        Enable < Length: 4, height: 2, color: blue, area ()>;
        // specifying wanted properties.
        Disable < angels: rounded, line-style: Straight >;
        // specifying unwanted properties.
}
```

- Interpretation process (selective inheritance algorithm) will produce the following genotype:

```
Rectangle1 <Polygon>
{
Enabled < EnabOrg = height: 4, height: 4, width=2, width=2,
                EnabDef= color: blue, line= dashed...
                EnabFun= area= h* w, circumference= 2(w+h)...>
// list of the Enabled properties.
Disabled <…………………> // list of the Disabled properties.
}
```

# CHAPTER FIVE

# "IS-A" VS. "COMPOSED BY" INHERITANCE MODEL AND COMPLEXITY

There is no known algorithm for “Is-a” based selective inheritance. So, it is useless to compute the complexity of our “Composed by” based selective inheritance. But, we can compare the complexity of “Is-a” hierarchy model with that of “Composed by” modeled by our approach.

In the following, we start by evaluating our “Composed by” inheritance model relatively to the” Is-a” one. We present the evaluation of the two models based on some standard complexity metrics. We end by concluding in the possibility of combining the two approaches “Is-a” and “composed by” in a single one.

### 5.1. "Is-a" Vs. “Composed by” Inheritance Models

As we know, current Inheritance concepts used in OOP and all current selective inheritance approaches are working under the "is-a" hierarchy model.

This hierarchal model has solved many issues, but on the other hand, led to the emergence of problems that needed solutions. Some of these problems were resolved in a holistic, but some others were resolved partially or have been resolved to reach the nearest satisfactory result.

For reducing some of the hierarchal model problems, several approaches proposed the selective inheritance. Some of these approaches helped in solving certain issues. But all the current selective inheritance approaches have been only applied on the inheritance "is-a" hierarchy model.

None of current approaches have worked on the nested class's model, although it supports inheritance, but is better in *organization* and *protection* than the inheritance hierarchy model. Where, (B.Eckel, 2006) says that with inner classes we have these additional features:

- Defining inner classes in an outer class may reduce the total number of outer classes in a software application.
- The inner class can have multiple instances, each with its own state information that is independent of the information in the outer-class object.

- In a single outer class you can have several inner classes, each of which implement the same interface or inherit from the same class in a different way.
- The point of creation of the inner-class object is not tied to the creation of the outer-class object.
- There is no potentially confusing “is-a” relationship with the inner class; it’s a separate entity.

So, our work has led us to take advantage(s) of the selective inheritance and the nested classes. As mentioned, our approach aims to enhance the inheritance hierarchy to the Genome model by merging all classes associated with the same aspect into one class. As known, *nested classes* concept is not new, but the new in our proposal suggests that after merging those classes, all their features will be classified according to control rules that ensures a correct classification where each class attribute contains all its possible alternatives. So, an object can select one or more alternatives from each wanted attribute, this selection process is also governed by interpretation rules that ensure a coherent selection. It is important to remind that classified classes inside the Genome have no inheritance relationships between each other. By this proposal, all concepts used in “Is-a” hierarchy model can be eliminated.

- The object will be able to inherit more than once from a specific class.
- *Polymorphism, Method-Overriding* and *Method-Overloading* concepts can be eliminated and replaced by *Alternatives* concept.
- *Circle-Ellipse* problem will be solved, because both the circle and ellipse objects will explicitly inherit from the same class, *Shapes.*
- The object will inherit selectively, which means, it will only have needed attributes / methods from *only* needed classes. This will solve the exponentially increased ancestors’ problem.

Besides that, this will make the designing easier, where the designer will not be forced to be aware of each class implementation; he just must identify the needed properties and the correct Genome (through the Genotyping Program), and the process of producing a coherent object will be finished by itself (through the Interpretation Process).

## 5.2 Complexity Measurements

The higher level of complexity requires more efforts in maintaining the software. In our work we will concern in classes, where, in software estimation and maintenance, the normal class complexity was claimed to be measured in term of the number of lines in code (LOC) which means the size of class, but there is no consensus on the idea that high class size is necessarily resulting in class complexity. Other metrics for complexity of classes is the structural and functional relationship among class elements (S.Tee, 2009):

- It should be noted that the degree of functional complexity is higher in the event that more class element interaction is found in a class.
- Structural complexity is done by using the *UML* representation.

### Inheritance hierarchy

Inheritance is claimed to reduce the amount of software maintenance and to ease testing. But some researches indicated that a system not using inheritance is better for understandability and maintainability than a system with inheritance (F.Sheldon et al, 2002). Because the inheritance is a hierarchy model, many metrics were investigated to measure the complexity. Some of these metrics are:

- *The depth of inheritance tree (DIT),* which in other words means, the number of ancestors that can affect a class. By this metric, it was agreed that the deeper the hierarchy, the better reusability of classes, but the higher the coupling between classes making it harder to maintain the system. For that, designers tend to keep the inheritance hierarchy shallow.
- *The number of subclasses that inherit methods from super-class (NOC).* Where, the greater number of subclasses, the greater ability to reuse, but the potential for improper abstraction for the super-class.
- *The number of ancestor classes (NAC)* from which a class inherits in the hierarchy. This metric was a developed to the *DIT* metric.
- *Weighted methods per class* (WMC), coupling between object classes, response for class, lack of cohesion in methods, and metrics for maintainability and understandability are also different metrics used to in complexity measurements.

Also, the more super classes your subclass inherits from, the more maintenance you are likely to perform. If one of the superclasses happens to change, the sub class may have to change as well.

So, while there is no single metric to measure the quality for a program that is using inheritance hierarchy, (F.Sheldon et al, 2002) has developed two simple metrics that measure the understandability and modifiability for class inheritance hierarchy.

If we applied the conventional metrics on the case study that we have in chapter two, see Figure (2-3), taking into account that this case study is a simplified example of what the Shapes tree can be in fact, we deduce the following:

- By applying *DIT* metric:

DIT (Shapes) = 0,

DIT (Curved) = DIT (Polygons) = 1,

DIT (Stereophonic-Curved) = DIT (Flat-Curved) = (Flat-Polygon) = DIT (Stereophonic-Polygon) = 2,

DIT (Parallelogram) = DIT (Non- Parallelogram) = DIT (Cone) = DIT (Cylinder) =3,

DIT (Dome) = DIT (Square) = DIT (Rectangle) = DIT (Hexagonal) = 4.

- By applying *NOC* metric:

NOC (Shapes) = 2,

NOC (Curved) = NOC (Polygons) = NOC (Flat-Polygon) = NOC (Stereophonic-Polygon) = NOC (Flat-Curved) = NOC (Parallelogram) = 2,

NOC (Stereophonic-Curved) = NOC (Non- Parallelogram) = 1,

NOC (Dome) = NOC (Square) = NOC (Rectangle) = NOC (Hexagonal) = NOC (Cone) = NOC (Cylinder) = 0.

- By applying *NAC* metric:

NAC (Shapes) = 0,

NAC (Curved) = NAC (Polygons) = 1,

NAC (Stereophonic-Curved) = NAC (Flat-Curved) = NAC (Flat-Polygon) = NAC (Stereophonic-Polygon) = 2, NAC (Parallelogram) = NAC (Non- Parallelogram) = 3,

NAC (Square) = NAC (Rectangle) = NAC (Hexagonal) = 4

NAC (Cone) = NAC (Cylinder) = 5,

NAC (Dome) = 6.

These metrics do not compute the total complexity for the hierarchy tree; they compute the complexity for each class separately. So, as mentioned, (F.Sheldon et al, 2002) extended new two metrics for maintainability (understandability and modifiability) of inheritance "directed acyclic graph" (DAG), where two functions are mainly used:

- *PRED* (i): the total number of predecessors of node (class) i,
- *SUCC* (j): the total number of successors of node (class) j.

Now, by applying the average degree of understandability (AU) which is defined by:

$$\text{Understandability (U) of class } (C_i) = \text{PRED } (C_i) + 1$$

$$\text{AU} = \left( \sum_{i=1}^{t} (PRED(Ci) + 1) \right) / \text{t}$$

Where, t is the total number of classes in the class inheritance DAG.

U (Shapes) = 1,
U (Curved) = U (Polygons) = 2,
U (Flat-Polygon) = U (Stereophonic-Polygon) = U (Flat-Curved) = 3,
U (Stereophonic-Curved) = 3,
U (Parallelogram) = U (Non- Parallelogram) = 4,
U (Dome) = 7,
U (Square) = U (Rectangle) = U (Hexagonal) = 5,
U (Cone) = U (Cylinder) =6.
AU= (1+2+2+3+3+3+3+4+4+7+5+5+5+6+6) /15 = 59/15 = 3.93

And by applying the average degree of modifiability (AM) which is defined by:

$$\text{Modifiability (M) of class } (C_i) = \text{U } (C_i) + \text{SUCC } (C_i) / 2$$

$$\text{AM} = \text{AU} + \left( \sum_{i=1}^{t} (SUCC(Ci) / 2) \right) / \text{t}$$

SUCC (Shapes)/2 = 14/2 = 7<br>
SUCC (Curved)/2 = 3/2 =1.5<br>
SUCC (Polygons)/2 = 9/2 = 4.5<br>
SUCC (Flat-Polygon)/2 = 5/2 =2.5<br>
SUCC (Stereophonic-Polygon)/2=2/2=1<br>
SUCC (Flat-Curved) /2=2/2 =1<br>
SUCC (Stereophonic-Curved)/2 = ½ = 0.5<br>
SUCC (Parallelogram)/2 = 3/2 =1.5<br>
SUCC (Non- Parallelogram)/2 = ½ =0.5<br>
SUCC (Dome)/2 = 0/2 =0<br>
SUCC (Square)/2 =0/2 =0<br>
SUCC (Rectangle)/2 =0/2 =0<br>
SUCC (Hexagonal)/2 = 0/2 =0<br>
SUCC (Cone)/2 =0/2 =0<br>
SUCC (Cylinder)/2=.0/2 =0<br>
AM= (3.93) + (7+1.5+4.5+2.5+1+1+0.5+1.5+0.5) /15<br>
= (3.93) + (20/15) = 5.26

**Nested classes**

Several metrics were used to find the complexity for inner classes. One of them is the metric which was developed in (S.Tee et al, 2009) that measures the complexity from the perspective of breadth and depth of inner classes. The complexity (C) value for inner classes is derived from the sum of breadth (b) to depth (d) ratio of the classes, where:

$b_i$: is the number of classes at level i.

$d_i$: is the level i.

Complexity (C) = $\sum_{i=1}^{n}\frac{bi}{di}$, where n is the number of levels.

Figure (5-1) explains the meaning of breadth and depth of inner classes. In Figure (5-1/ a) we see that the outer class “*Class 1*” contains three inner classes “inner A, inner B and inner C” at the same breadth. These inner classes are defined at the same level of depth. But Figure(5-1/b) shows an outer class “*Class 2*” that contains immediate inner class “Inner D” that contains immediate inner class “Inner E” that also contains immediate inner class “Inner F”. We notice that the inner classes are defined at different level of depth.

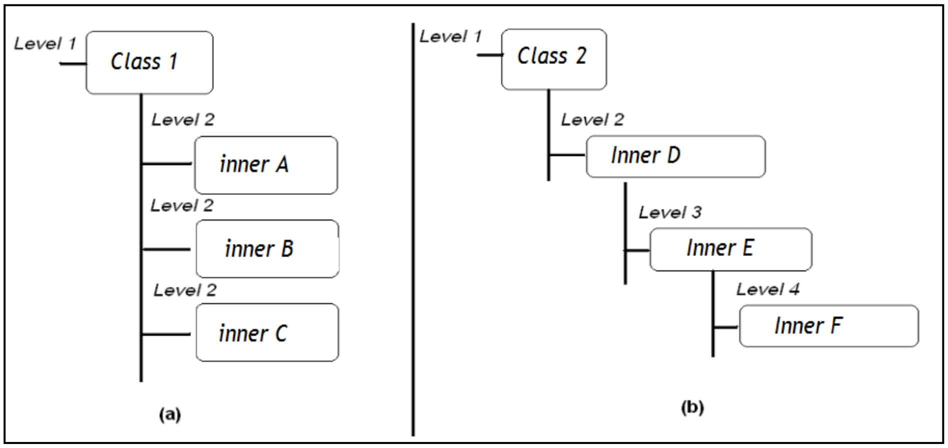


Figure (5-1): Inner Class Complexity from the Perspective of Breadth and Depth.

The structures of Class1 and Class2 may be as follows:

```
Class class1
{
  Class innerA {};

  Class innerB {};

  Class innerC {};
}
```

```
Class class2
{
  Class innerD
      {
        Class innerE
          {
             Class innerF {};
           };
        };
}
```

We want to clarify that in our case study we have taken a small portion of the Geometry shapes' full tree, and we have calculated the complexity for this portion that only produced the following six classes *Dome*, *Square*, *Rectangle*, *Hexagonal*, *Cone* and *Cylinder*. So, to be fair in our calculation, we will only consider the same portion after applying our approach on the case study as in Figure (5-2) which was deducted from Figure (4-7), we notice that we have an outer class " Geometric-Shapes " that contains two inner classes: *Polygons and Solids*. Each inner class does not inherit from any other inner class, and contains several inner classes; for our example *Polygons* class will contain *Quadrant and more-Quadrant*, *Solids* class will contain *No-Surface*s. So, all inner classes we gain are defined at multiple breadth and depth. Now, we will try to apply this metric on our approach:

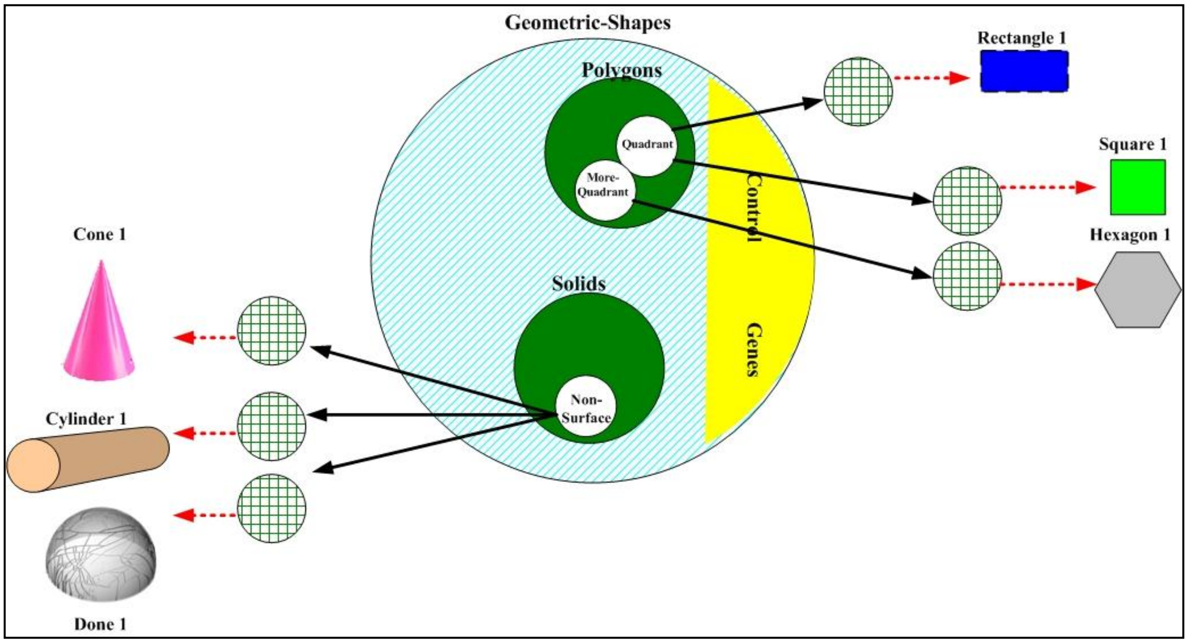


Figure (5-2): Deducting a Portion from Figure (4-7) that is Relevant to the Portion in the Case Study

Breadth (b1) at level (1): 1,

Depth (d1) at level (1): 1,

Breadth (b2) at level (2): 2,

Depth (d2) at level (2): 2,

Breadth (b3) at level (3): 3,

Depth (d3) at level (3): 3,

Then:

Complexity (C) = $\frac{b1}{d1}+\frac{b2}{d2}+\frac{b3}{d3} = \frac{1}{1}+\frac{2}{2}+\frac{3}{3} = 3$

When comparing between the two approaches complexity, we can see the big difference in calculations results for the benefit of our approach.

### 5.3. Combining between our Approach and the "is-a" Hierarchy Model

Some may ask why we have omitted the inheritance relations between inner classes, or in other words, why we can not combine the inheritance hierarchy inside our approach as shown in figure (5-3).

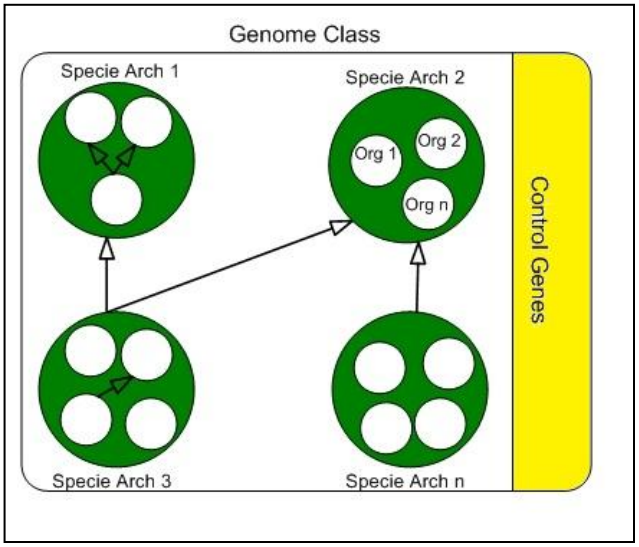


Figure (5-3): Combining "is-a" Model with “Composed by” Model.

In our work, doing that combination is not useful and not logical; where we classify the classes according to control rules, so that each class will contain its related inner classes and all its related properties, and each property has a set of alternatives, for that there will not be any necessary for using the “is-a” hierarchy. An aim of our work was to eliminate several problems that have been resulted from using the hierarchy such that conflict names, huge number of ancestors and super-classes and many other issues that are well known.

If we look at the case study implementation illustrated in figure (4-6) we see that the *Geometric-Shapes* contains *Curved*, *Polygons* and *Solids* where each one has its own inner classes, which makes the inheritance relation between them not logical because ,for instance, there is nothing to inherit from *Curved* into *Polygons* and vise versa. Also, if we took a close look inside the *Polygons* we will see several classifications: *Quadrant, More-Quadrant and less-Quadrant* classes, each class contains the associated properties, functions and behaviors that differ from the other classes. For instance, the Square has a different properties and functions from the Triangle, so it will gain its needed properties from the class *Quadrant*, wherefore; *Quadrant* does not inherit from any other class, and so on.

# CHAPTER SIX

# IMPLEMENTATION ISSUES, EVALUATION AND APPLICATION AREAS

In this chapter, we will present an implementation of our work using a language extension that produced in (S.Ghoul, 2011) work, we will evaluate our work according to several criteria, we will suggest several extensions to our work as a future work, and finally we will present some of the areas where our work may be applicable in.

## 6.1 Implementation Issues

Our approach helps in reducing the number of classes, instantiating different objects from a class without the need of adding any methods/ attributes after creating these objects to distinguish them from each other.

In the following, we will propose how a Genome and Genotype Program may be implemented in an object oriented language depending on the work (S.Ghoul, 2011).

**Genome Implementation**

```
Genome_Aspect_Name :( Genome Class= Configuration)
{
   Config: (Name=Dominant) {};

   Config: (Name= Specie-Name_j)
        {
        Archi:( Specie-Architecture-Name_i)
            {
            Require {
                  structure (st_1=<al_1,al_2…>, st_2< al_1,al_2…>,…);
                  function (st_1-fun=<al_1,al_2…>, st_2-fun< al_1,  al_2…>,…);
                  behavior (st_1-fun-be=<al_1,al_2…>, st_2-fun-be < al_1,al_2…>,…);
                        }; //End Require
             Imply{
                  structure (…);
                  function (…);
                  behavior (…);
                      }; // End Imply
             Exclude{
                  structure (…);
                  function (…);
                  behavior (…);
                           }; // End Exclude
                  }// End Archi: Architecture-Name_i
          }// End Config: Specie-Name_j

   Config: (Controls)
        {
        Cont:(Specie-Control){};
        Cont:(Architecture-Control){};
        Cont:(Structure-Control){};
        }; // End Config: Controls

}// End Genome Configuration
```

Where,

- Dominant: if no Specie-Name were selected in the genotype program, then the dominant Specie is assigned where it holds all its coherent attributes.
- Require: defines all the structures, functions, and behaviors with all their possible alternatives that must be imposed in the selected Specie/ Architecture.
- Imply: defines all the structures, functions, and behaviors with their possible alternatives, which may be implied by the imposed attributes.
- Exclude: Imply: defines all the structures, functions, and behaviors with their possible alternatives, which may be eliminated from the Specie/ Architecture by the imposed attributes.

**Genotype Implementation**

```
Function Genotype genotype-name ()
{
  Specie-Name=…;
  Specie-Architecture=…;
  Enable set={structure ($st_1$=<$al_j$>, $st_2$< $al_i$>,…);
              function ($st_1$-fun=<$al_k$,…>, $st_2$-fun< $al_m$, …>,…);
              behavior ($st_1$-fun-be=<$al_k$,…>, $st_2$-fun-be < $al_m$,…>,…);
             }
  Disable set={structure ($st_1$=<$al_j$>, $st_2$< $al_i$>,…);
              function ($st_1$-fun=<$al_k$,…>, $st_2$-fun< $al_m$, …>,…);
              behavior ($st_1$-fun-be=<$al_k$,…>, $st_2$-fun-be < $al_m$,…>,…);
             }

} // End Genotype
```

Where,

- Enable set: to define the attributes (structure, functions, and behavior) that wanted to be enabled.
- Disable set: to define the attributes (structure, functions, and behavior) that wanted to be disabled.

## 6.2 Evaluation Criteria

Since there are several approaches for selective inheritance concept, each approach, as we mentioned, has adopted it from a different point of view, and also, each approach has its defects. In our approach we tried to overcome these defects.

The comparison between these approaches, must take the following evaluation criteria into account:

- Genetic class: The approach possibility of grouping classes into one class, in order to reduce the number of classes.
- Selection at class level (non-inheritable traits): the approach possibility to allow the class to make some properties unable to be inherited.
- Properties classification: the approach possibility of grouping alternatives of the same property.
- Conflict situation resolution: the approach possibility of controlling selection process to prevent the selection of contradictory properties.
- Selection at object level: the approach possibility of allowing object to dynamically select wanted but coherent traits.
- Selection rules / control rules: does the approach provide rules that govern and control the selection process.
- Fully Bio-inspired: has the approach fully modeled the Genetics concepts which are the Genome, Genotype, and Phenotype and their control Genes. Where, several different Genotypes can be generated from the Genome, and several different Phenotypes can be instantiated from each Genotype. In computing world, the Genome offsets a "Software Database", the Genotypes offsets the "Views" that can be generated from the Software Database, and the Phenotype offsets the "Instances" from each View.

Table (6-1) shows a comparison (based on the above criteria) between our proposed approach and the previous studied ones.

As we can see, our approach achieves the previous criteria where:

- Genetic class: inspiring from Genome, our approach used a "Composed by" class that contains all classes related to specific aspect.
- Selection at class level (non-inheritable traits): each class has its control genes which are unable to be inherited.
- Properties classification: each property, function, and behavior has been classified, so that each one has a list of alternatives to select from it.
- Conflict situation resolution: control's genes at each class, control the selection process to prevent the selection of contradictory properties.

| Criteria<br>Approach | | Genetic class | Selection at class level | Properties classification | Conflict situation resolution | Selection at object level | Selection rules / control rules | Fully Bio-inspired |
|---|---|---|---|---|---|---|---|---|
| S.I at Class Level | | × | ✓ | × | × | × | × | × |
| S.I at Object Level | Non-Genetic | × | × | × | × | ✓ | × | × |
| | Genetic | ✓ | × | ✓ | ✓ | ✓ | × | × |
| Our Approach | | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |

Table (6-1): Comparison between our Approach and the Previous Approaches

- Selection at object level: inspired from Genotype, our approach allows the object to dynamically select wanted but coherent properties through the "Genotyping Program".
- Selection rules / Control rules: inspiring from the control's genes, our approach provides rules that govern and control the selection process through the "Interpretation Process".
- Fully Bio-inspired: to be a full genetics-based approach, our approach has fully modeled the genetics concepts which are the Genome "a Composed by class", Genotype "object with needed properties" and their control Genes, beside an Interpretation Rules "Selective Inheritance".

## 6.3 Application Areas

Our concepts are closer to real life than the concepts adopted in current OOP, the number of classes is reduced, the variation in the same class is supported, and the inheritance is more powerful and practical than it is now in the Conventional OOP. This approach is needed in any application that significantly uses the inheritance, so, it can be applied to select the best object parameters in any object-oriented computer environment. As an example, in hardware field, (the micro-architecture is usually designed and tested with the aid of a software simulator) where designing, testing, and producing a new computer processor is complex, (J.Bastian et al, 2005) has proposed a specification of INTEL IA-32 using an architecture description language that selectively pre-determined the parent with the *best fit* to the object.

Also our approach may be applied in several works: Software Process Modeling, Software Reengineering, Software Reuse and relational databases. The results may be original and promising.

# CHAPTER SEVEN

# CONCLUSION AND FUTURE WORKS

## 7.1 Conclusion

Through the studying of the inheritance used in current OO, we found that it does not mimic the natural (real life) inheritance process as was claimed. In our approach, to eliminate all current OO inheritance's problems, we used the Genetics concepts to model a selective inheritance that is closer to our natural life. In our model, an object can be created holding only desired and necessary properties and methods. This approach is young and new, it will be formalized and evaluated when it is largely accepted.

## 7.2 Future Works

Our work can be extended and developed in future to:

- Accidents study (error handling).
- To model the other type of genotyping; which is a genetic interaction with the environment. It is based on merging a given genome with another introduced from the environment.
- Mutation and genotype.
- Work in depth inside the gene where there are *silent* and *active* genes that may work an essential role in the genotype evolution process.
- Evaluating our model according to the memory requirements, time needed to initialize an instance, and tightly coupled feature.

# REFERENCE

Al-Ahmed,W and Steegmans, E, (1999). Improving Support for Specialization Inheritance. In Journal of Object-Oriented Programming, January 1999, pp.29-36.

Anban Pillay, (2007). Object Oriented Programming using Java . Adapted from Introduction to Programming Using Java. School of Computer Science,University of KwaZulu-Natal, February, 2007.

Bruce Eckel, (2006). Thinking in Java (4th Edition) book.

Dov Dori, Erez Tatcher, (1994). Selective multiple inheritance. IEEE software, 1994.

F.Sheldon, Kshamta Jerath and Hong Chung, (2002). Metrics for Maintainability of Class Inheritance Hierarchies. Journal of Software Maintenance and Evolution: Research and Practice, Ref:SMR249/24343ae, 2002.

Joe Blaylock, (2008). Luis Rocha's Agent-Based Model of Genotype Editing. January, 2008.

J. Bastian and S. Õnder, (2005). Specification of Intel IA-32 using an Architecture Description Language, IFIP International Federation for Information Processing, 2005, Volume 176, Architecture Description Languages, Pages 151-166.

J. Meslati, Said Ghoul, (1997). Semantic Classification:A Genetic Approach To Classification In Object-Oriented Models. 1997.

NeedMarkku Sakkinen, (2005). Wishes for object-oriented languages. Invited paper at LMO 2005, Bern, 9 March 2005.

S. Ghoul, (2010). Bio-inspired Systems-An Integrated Model. Misc2010-Constantine, 30-31 May, 2010.

S.Ghoul, (2011). Supporting Aspect-Oriented Paradigm by Bio-Inspired Concepts. Fourth International Symposium on Information & Communication Technology, IEEE, 2011.

Sim Hui Tee, (2009). Developing a Complexity Metric for Inner Classes. Journal of Theoretical and Applied Information Technology, 2005 - 2009 JATIT.

Stephan Herrmann, (2005). Programming with Roles in ObjectTeams/Java. American Association for Artificial Intelligence, 2005.

Steven te Brinke, (2007). First-order function Dispatch in a Java-like programming language. Master of Science dissertation, University of Twente, January, 2011.

Tim Otter, (2005). Genotype, Phenotype and ontogeny. GECCO'05,june 25-29,2005, Washington, DC, USA.

Timm Owen Martin, (2004). Selective inheritance of object parameters in object-oriented computer environment. U.S patent, march, 2004.

Tomas Oplustil, (2002). Inheritance of SOFA Components. Master Thesis, Masaryk University, 2002.

Will Braynen, Simon Angus, Paul Dwyer, Mollie Poynton, Alejandro Balbin, and Risi Kondor, (2007) . Genotype or Phenotype? The conflation of two concepts in evolutionary agent-based modeling. supported by the Santa Fe Institute through NSF Grant No. 0200500 entitled "A Broad Research Program in the Sciences of Complexity.”, 2007.

Wikipedia.org/wiki/Circle-ellipse_problem, ( 2012).

# APPENDIX

The following is a full algorithm for each function:

```
Void Test-In- Specie-Control-Genes (input specie-name; output specie-architecture);
{
  If specie-name = null
      Specie-name= AreDdominant specie ();
  Choose specie-architecture (specie-name);
}

 Void Visit- species-Architecture (input specie-arch; output species- architecture)
  {
        Species- architecture is object;
        Read specie-arch;
        Choose the correct species- architecture;
  }

Void Test-In-Architecture-Control-Genes (input  specie-arch, species-Architecture;
output arc-genes);
{
  While species-Architecture not empty ()
  {
    AreImplidInArchitecture(specie-arch);
      Put in arc-genes;
    AreExcludedFromArchitecture(specie-arch);
      Remove from arc-genes;
    if  organ & organ-definition &function & behavior = null
      AreDefaultInArchitecture(specie-arch);
      Put in arc-genes;
  }
  Output arc-genes;
}

Void Select-coherent-organs(input arc-gennes,EnabOrg, Disaborg)
{
   while Enaborg & Disaborg  not empty()
    {
      read organ
      Visit Species-Architecture-Genes (arc-genes,Enaborg, Disaorg)
     }
  Test-In-Organs-Control-Genes (Enaborg, Disaorg; coh-org);
  Add coh-org to Enaborg;
}

Void select-ontogenesis(input Genome, EnabOrg, Enabdef, Enabfun, Enabbeh,
Disaborg, Disabdef, Disabfun, Disabbeh; output genotype)
{
  While Enaborg not empty()
    {
        Read organ
```

```
            While Enabdef & Enabdef & Enabbeh not empty()
            {
                Read organ-definition, function,behavior;
                Visit Organ$_i$ (Enabdef, Enabfun, Enabbeh);
                If Test-In-POE-Control-Genes (Enabdef, Enabfun, Enabbeh) is true ;
                  Visit-ontogenesis-Genes (Enabdef, Enabfun, Enabbeh);
            }

              Test-In-Ontogenesis-Control-Genes (Enabdef, Enabfun, Enabbeh;coh-alt);
              While coh-alt not empty()
                 {
                    Visit-Organ-Definition-Genes (organ-definitions; d-alternatives);
                    Visit-Organ-Function-Genes (functions, f-m-alternatives, f-s-
                            alternatives);
                    Visit-Organ-Behavior-Genes (behaviors, b-m-alternatives, b-s-
                            alternatives);
                 }
          }
                    AreGenotypeGenes();
   }

void Visit Species-Architecture-Genes (input arc-genes, Enaborg, Disaorg )
{
   Read arc-genes, Enaborg, Disaorg;
}

void Test-In-Organs-Control-Genes (input arc-genes, Enaborg, Disaorg; output coh-
org)
{
  while arc-genes not empty()
  {
     PerformSameFunctionsOrgans(Enaborg)
     AreImpliedOrgans(Enaborg)
        Put in Enaborg;
     AreImpliedOrgans(Disaborg)
        Remove from Enaborg;
      AreExcludeOrgan(Enaborg);
       Remove from Enaborg
     AreExcludeOrgan(Disaborg);
        Remove from Enaborg
   }
Coh-org= Enaborg;
}


Void Visit Organ$_i$ (Enabdef, Enabfun, Enabbeh)
{
    read Enabdef, Enabfun, Enabbeh;
}
```

```
Test-In-POE-Control-Genes (Enabdef, Enabfun, Enabbeh)
{
  if AreRelatedToConstruction (Enabdef, Enabfun, Enabbeh)
     Then true;
}

Visit-ontogenesis-Genes (input Enabdef, Enabfun, Enabbeh; output ontogenesis-gens)
{
  read Enabdef, Enabfun, Enabbeh;
}

Test-In-Ontogenesis-Control-Genes (input ontogenesis-gens, Enabdef, Enabfun,
Enabbeh, Disabdef, Disabfun, Disabbeh;coh-alt)
{
  while ontogenesis-gens not empty()
   {
     While Enabdef& Enabfun & Enabbeh not empty()

Applying Rules:
    AreRelatedToAspect();
    AreImplyed()
    {
       definition in Enabdef → enable definition in ontogenesis-gens;
       function in Enabfun → enable function in ontogenesis-gens;
       behavior in Enabbeh → enable behavior in ontogenesis-gens;
            put in coh-alt;
       definition in Disabdef → Disable definition in ontogenesis-gens;
       function in Disabfun → Disable function in ontogenesis-gens;
      behavior in Disabbeh → Disable behavior in ontogenesis-gens;
          remove from coh-alt;
    }

    AreExclusive()
    {
        definition in Enabdef <> enable definition in ontogenesis-gens;
         function in Enabfun <> enable function in ontogenesis-gens;
         behavior in Enabbeh <> enable behavior in ontogenesis-gens;
             remove from coh-alt;
         definition in Disabdef <> Disable definition in ontogenesis-gens;
         function in Disabfun  <> Disable function in ontogenesis-gens;
        behavior in Disabbeh  <> Disable behavior in ontogenesis-gens;
            remove from coh-alt;
      }


if two or more contradictory definitions  then AreDominant();
if two or more contradictory functions  then AreDominant();
if two or more contradictory behaviors  then AreDominant();

AreGenotypeGenes()
```

```
{
   Genotype= Set of { specie-arch; arc-genes; Enaborg (Enabdef (d-alternative);
                                                                      Enabfun(f-m-alternatives, f-s-alternatives);
                                                                     Enbbeh(b-m-alternatives, b-s-alternatives))
                                                              }
}

Visit-Organ-Definition-Genes (input coh-alt; output  d-alternatives)
{
   while coh-alt not empty()
     If definition-gene then select on alternative from definition alternative;
       Put in d-alternative;
}

Visit-Organ-Function-Genes (input coh-alt; output  f-m-alternatives, f-s-alternatives)
{
   while coh-alt not empty()
        If function-gene then
            {
            select on alternative from Major-function-alternative;
                Put in F-m-alternative;
            select on alternative from Sub-function-alternative;
                Put in F-s-alternative;
            }
}

Visit-Organ-Behavior-Genes (input coh-alt; output b-m-alternatives, b-s-alternatives)
{
   while coh-alt not empty()
      If behavior-gene then
           {
            select on alternative from Major- behavior -alternative;
                 Put in b-m-alternative;
             select one alternative from Sub- behavior -alternative;
                 Put in b-s-alternative;
            }
}
```

# ملخص

مفهوم الميراث التقليدي المعتمد في البرمجة الموجهة الحالية (OOP) والتي تعمل على نموذج التسلسل الهرمي "is-a" ، لديه بعض العيوب . وحيث (OOP) تحاول أن تكون أكثر قرباً من واقع الحياة، فإنها لا تزال بعيدة كل البعد عن مبادئ علم الوراثة.

الوراثة تعني أن الفئة الفرعية (child class) يمكن أن ترث وتحصل على كل ما هو عام في الفئة الأصل (parent class) تلقائيا .هذه العملية قامت بحل العديد من المشاكل، لكنها لا تحاكي ما يحدث حقا في حياتنا، حيث أن كل كائن (object) يستطيع الحصول فقط على الصفات المطلوبة من الفئة الأصل .هذا يعني أن مفهوم الوراثة التقليدي ليس انتقائي ويولد كائنات متطابقة.

وبسبب أن مفهوم الوراثة التقليدي يعمل على نموذج التسلسل الهرمي "is-a" ، فإن الأعمال التي قدمت في مفهوم الميراث الانتقائي أيضاً تمت على هذا النموذج الهرمي.

أدى الاستيحاء من الجينات في عملنا إلى مفهوم وراثة انتقائي مبني على نموذج ”Composed by“ بدلا من نموذج "is-a" . حيث في علم الأحياء، يتم اختيار الصفات المطلوبة من "الجينوم" الذي يحمل كل السمات المخصصة لجانب معين ، وتصنف تلك الصفات إلى طبقات عدة وفقا لعلاقة ”Composed by“.

بعد التقييم، تبين أن الوراثة الانتقائية التي تعمل على النموذج ”Composed by“ هي أفضل من الوراثة الانتقائية المبنية على نموذج التسلسل الهرمي "is-a".

# منـهجية مبنية على علم الجينات لنمذجة الوراثة

بواســطة
دارين موسى محمد حمودة

بإشــراف
أ.د. سـعيد الغول

قدمت هذه الرسالة استكمالاً لـمـتطلبات الحصول على درجة
الــمـاجـستير في عـلـم الـحـاسـوب

عمـادة البحث العلمي والدراسات العليا
جامعة فيلادلفيا

آذار 2012