A Bio-Inspired Objects Evolution Methodology

By

Enas Tawfiq Sa'ad Al-Naffar

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

May 2012

جامعة فيلادلفيا

نموذج تفويض

أنا إيناس توفيق النفار، أفوض جامعة فيلادلفيا بتزويد نسخ من رسالتي للمكتبات أو المؤسسات أوالهيئات أوالأشخاص عند طلبها.

التوقيع :
التاريخ :

Philadelphia University
Authorization Form

I am, Enas Tawfiq Sa'ad Al-Naffar, authorize Philadelphia University to supply copies of my thesis to libraries or establishments or individuals upon request.

Signature:
Date:

A Bio-Inspired Objects Evolution Methodology

By
Enas Tawfiq Sa'ad Al-Naffar

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

May 2012

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

Examination Committee	**Signature**	
Dr. Academic Rank:	, Chairman.	_ _ _ _ _ _ _ _ _ _ _ _
Dr. Academic Rank:	, Member.	_ _ _ _ _ _ _ _ _ _ _ _
Dr. Academic Rank:	, Member.	_ _ _ _ _ _ _ _ _ _ _ _
Dr. Academic Rank: ( )	, External Member.	_ _ _ _ _ _ _ _ _ _ _ _

Dedication

I dedicate this Thesis to my wonderful family; my loving and giving parents in particular; for their endless love and unconditional support each step of the way. Thank you Mum and Dad, without you, I would never have been able to accomplish any achievement. To my sisters and my brothers; for their patience, help and support throughout the years. I am grateful for all the things you have done for me.

Also, I dedicate it to my amazing friends, who were always there for me, who encouraged me and did their best to support me in every way they could.
To my Supervisor Prof. Said Ghoul, who motivated me to join the Master's program and helped me in every possible way.

Last but not least, I would like to dedicate my Thesis to those of my homeland who are struggling to have a normal life, and might not have had the same opportunities that I had.

Enas Al- Naffar

Acknowledgment

I would sincerely like to thank my Supervisor Prof. Said Ghoul, for his help, patience and unlimited support during the years.
He has always been ready to help; his constant encouragement and his valuable advice have lit my way in the darkness and helped me find the right path. He has not only been a supervisor to me, but more like a father, a friend and an inspiring teacher.

My sincere gratitude for my devoted Doctors and Teachers at Philadelphia University; who gave us the best they had. I really appreciate their effort and time trying to put me and my colleagues on the right track.
Above all, I thank God for his blessings; for giving me strength, patience and desire.

To everyone helped me during my entire study and throughout this thesis...Thank you.

Enas Al-Naffar

Table of Contents

Subject	Page
Authorization Form	ii
Title	iii
Examination Committee	iv
Dedication	v
Acknowledgement	vi
Table of Contents	vii
List of Figures and Tables	viii
List of Abbreviations	ix
Abstract	x
Chapter One: Introduction	1
1.1 Problem statement	2
1.2 State of the art	2
1.3 Motivation	3
1.4 Contribution	4
1.5 Thesis layout	6
Chapter Two: A Case Study	7
Chapter Three: Literature Review	12
3.1 Object Evolution Approaches	13
3.2 Fault Tolerance Approaches	20
3.3 Literature Review Conclusion	21
Chapter Four: A Genetics-Based Methodology to Object Evolution	22
4.1 Genetics-Based Evolution Methodology	23
4.2 Genetics-Based Evolution Concepts	24
4.3 Genetics-Based Evolution Scope	26
4.4 Tolerating Evolution Accidents	34
Chapter Five: Evaluation: Implementation Issues, Application Area and Comparison	37
5.1 Implementation Issues	38
5.2 Application Area	39
5.3 Comparison	40
Chapter Six: Conclusion and Future Works	43
6.1 Conclusion	44
6.2 Future Works	44
References	45

List of Figures and Tables

Figure Number	Figure Title	Page
Figure (1-1)	Object (Phenotype) evolution example.	5
Figure (2-1)	Hierarchy of List example, in a traditional approach - using UML notations.	8
Figure (3-1)	Classes and objects hierarchy following class evolution and schema evolution approaches.	16
Figure (3-2)	Our case study using dynamic roles.	18
Figure (3-3)	Our case study using ORM.	18
Figure (3-4)	The POE model. Partitioning the space of bio-inspired systems along three axes: Phylogeny, Ontogeny, and Epigenesis concepts.	19
Figure (4-1)	Genetics-based evolution methodology.	23
Figure (4-2)	Genetic evolution process.	25
Figure (4-3)	GEC algorithm, in pseudo code.	27
Figure (4-4)	A genetic lifecycle (GEProc) of an object of Static_List	28
Figure (4-5)	Behavior Q-Beh0 of an object in StQueue state.	29
Figure (4-6)	A genetic lifecycle of an object evolving between Static_List and Dynamic_List genotypes by Metamorphosis programs.	31
Figure (4-7)	Accidents tolerance algorithm, in pseudo code.	36
Figure (5-1)	An extension to an OOPL.	39
Figure (5-2)	A life cycle of a microscopic entity.	40
Table (5-1)	A Comparison between object evolution approaches.	41

List of Abbreviations

GEC	Genetics-based Evolution Control
GEProc	Genetic Evolution Process
GEProg	Genetic Evolution Program
GER	Genetic Evolution Relations
GUI	Graphical User Interface
OOM	Object-Oriented Model
OOP	Object-Oriented Paradigm
OOPL	Object-Oriented Programming Language
ORM	Object Role Modeling

Abstract

One of the major issues of object-oriented programming languages is the lack of reclassification mechanisms. Reclassification allows an object to change its class at runtime. Object reclassification is desired in applications whose entities need to change dynamically at runtime.

The previous approaches dealt with object reclassification in different ways. But none had approached the notion of real entities evolution, where objects can evolve while belonging to the same class. This had led to a large number of classes most of the times; making the design and the implementation of the intended system more complex.

The desired evolution of objects should allow objects to evolve automatically at runtime, without changing their classes membership. In this work, we present a new approach for objects evolution; inspired by some genetics concepts. Objects that belong to the same class can change their structure, functions and behaviors at run time automatically, while keeping their membership to the same class. A slight environmental influence on the proposed evolution process is introduced.

Keywords - Bio-inspired model, evolution, genome, genotype, phenotype.

CHAPTER ONE

INTRODUCTION

Object evolution is one of the issues that has been addressed recently in the area of object-oriented programming. There have been some works conducted in this filed. In this work, we present a new approach to be followed in the area of object evolution.

1.1 Problem Statement

Object-oriented paradigm (OOP) was invented for the purpose of physical modeling. However, not everyone agrees that there is a direct mapping between real world concepts and OOP, since an object-oriented program is considered in many cases as a model of only some parts of the world. Bertrand Meyer argues in *Object-Oriented Software Construction* (B.Meyer, 2000) that a program is not a model of the world but a model of some parts of the world. Class-typed programming languages do not usually provide reclassification mechanisms that allow objects to change their classes membership. Lack of reclassification primitives has long been recognized as a practical limitation of object-oriented programming (D.Ancona et al, 2007).

Many applications, such as graphical user interface (GUI) applications, games, social databases, etc., require dynamic changes of objects. It is very common to change the value of their attributes; this is done by Setter functions (mutators). But what we really need is beyond changing the values of the attributes; we need to be able to change the attributes themselves; to add and/or remove some attributes. We also need to be able to change the operations and the behavior of the objects over their running time. In other words, what we really need is an object that *evolves automatically at runtime*.

The previous approaches dealt with objects evolution in different ways. But none had approached the notion of real entities evolution (phenotype evolution). This had led to a large number of classes most of the times; making the design and the implementation of the intended system more complex.

1.2 State Of the Art

There have been some attempts to find a solution for the issue of objects evolution which refers to changes in objects at runtime. Some previous research works focused on the area of *re-classification*, where an object may change its class at runtime. This has been undertaken through languages like Fickle (D.Ancona et al, 2007). There have been some works that dealt with *object evolution* as a restriction of *dynamic object reclassification* (T.Cohen and J.Gily, 2009*)*, where evolution here, allows an object to gain, but never lose capabilities.

Other works dealt with *Class evolution* (E.Johnsen et al, 2009) which is supporting an arbitrary set of changes to classes, which may have large number of persistent objects. In object-oriented persistent platforms, the changes conducted on classes require modifying existing objects; their contents and behaviors. Some works have been conducted in this area, in order to change the classes' instances to conform to the new descriptions. The work conducted in (E.Johnsen et al, 2009), presents a language that can introduce new functionality and interfaces for classes. In (M.Piccioni et al, 2011), refactoring units and object transformers are used to create an instance of the new class from a serialized instance of the old class

There have been different approaches for extending traditional object model with role mechanisms. Roles define extra properties which are added to objects. During its lifetime, an object may adopt and abandon roles. The approach in (A.Jodloowski et al, 2004) and (D.Stein et al, 2005) assumes that role arranges a hierarchy similar to classes. An object can have many roles which can be added/removed at run time. A role has its own attributes and behavior. It dynamically "imports" attributes (values) and behavior from its super-roles, in particular, from its parent object. In (A.Caetano et al, 2005) and (T.Halpin, 2010), the role has been presented as a static entity (structure without operations).

In (S.Ghoul, 2010), a bio-inspired objects evolution principle is presented based on the experimentation of (D.Meslati and S.Ghoul, 2005) that dealt with software evolution using biological concepts. In (D.Meslati and S.Ghoul, 2005) a software system includes a structural, behavioral and an ontogenetic dimension. All changes undergone by a software system are considered as its *ontogenetic* dimension. The model of a software system consists of a *phenotype* and a *genome*. The *phenotype* (object) captures its structural and behavioral dimensions from the genome; while the *genome* (class hierarchy) captures all changes that shape the system to keep it conform to the changing environment and requirements.

1.3 Motivation

Several needs, concerning object evolution, come out from studying previous works:

- We need objects that can change their structures, operations and behavior at run time dynamically, without having to change their class membership.
- We need a well-defined model for this kind of evolution.
- We need an object-oriented programming language extension to support this model.

- We need to be able to handle some kinds of accidents that may occur in the environment during objects evolution.

1.4 Contribution

This work aims to solve the problem of object evolution, inspired by real entities (phenotype) evolution; reducing the gap between real world concepts and computing concepts. So, an object may evolve automatically at runtime while keeping its membership to the same class. Our work is built on previous works conducted in (D.Meslati and S.Ghoul, 2005) and (S.Ghoul, 2010). These two precedent works present only a general evolution philosophy. In this work, we complete them by modeling the process of object (phenotype) evolution. We are concerned with *ontogenesis* aspect. The *phenotype* is considered as an instance of a specific class, genotype (D.Hammodeh, 2012). A phenotype can change its structure, operations and behavior without affecting its genotype (the genotype remains the same as long as no mutation process takes place). Just like a phenotype, once an object is created from a specific class; it can evolve automatically during its lifetime, with the ability to handle some kinds of environmental accidents.

Our study is limited, in this work, to a static (predefined) genetic evolution plan, with a slight environmental influence, that includes some common kinds of accidents. During objects lifetime, various types of accidents may occur in the environment and affect the evolution process and objects, as well. Accidents should be detected first, and then the damage should be assessed, to determine what parts of the object are affected by the accident (i.e. an accident may occur and could cause a damage in object's structures). Different mechanisms can be used to remove the damage and restore the object to a safe state (I.Sommerville, 2011).

To illustrate the need for a new approach to be followed in the area of objects evolution, consider a domain that represents a group of humans and their interactions. A class Human may have attributes like name, age, gender, etc. The values of these attributes could simply be changed at runtime. We should take into consideration that in real world, Human is not a static entity, as illustrated in Figure (1-1). Human starts as a Child with specific attributes such as name, height, weight, birth date. This Child grows up to become a School_student. This student will have extra attributes like school_name, hobby…etc. The student grows up into an Undergraduate_student, which will have extra attributes and so on. Note that during the growth process, not only the attributes are changed, but also the operations and the

behaviors are changed; operations may be either gained or lost. For example the operation get_HighSchoolAvg will be gained when the Child grows up (evolves) into a Student. Also note, that some attributes may be lost as the object evolves.

Figure (1-1): Object (Phenotype) evolution example

This kind of domains is traditionally modeled using many classes, i.e. Human, Child, Student, Undergraduate_student, Employee...etc., with an inheritance relationship among some classes, i.e., Human is a super-class while the rest of classes are sub-classes. To capture the process of human evolution, an object has to change its membership between two classes. For example an object of type Child may change its membership to belong to Student class. This is not practical; since in real world, the growth (evolution) of an object (phenotype) does not actually change its class.

Our approach solves this problem differently; by only having one composed class. Instances of this class can evolve automatically at run time, while remaining in the same class. Instances can change (gain/lose) their attributes, functions and behaviors. With this approach, we aim to enhance the area of objects evolution, by applying some genetics concepts that are related to phenotype growth. This enhancement will result in a well-defined model for objects evolution.

In this work, we introduce the following concepts:

- A genetic evolution process that determines a predefined lifecycle of an object (our work is limited to predefined evolution).
- A genetic evolution program that determines the acquired structures and functions, as well as the lost ones, for each evolution.
- Control rules that interpret the genetic evolution program, according to pre-defined genetic relations, into an object.
- Accidents tolerance to handle some accidents that occur during objects evolution.
- An object-oriented programming language extension to support our model.

1.5 Thesis Layout

In the following, we present in Chapter Two a case study that will be used along this thesis. Chapter Three is devoted to some significant approaches that are followed in the area of object evolution. In Chapter Four, we will present our contribution, a Genetics-Based Methodology for Object Evolution. An evaluation of our work is presented in Chapter Five. In Chapter Six, we finalize with future works and a conclusion of our work.

CHAPTER TWO

A CASE STUDY

In this chapter, we introduce a simple and a common case study that will be used throughout this thesis. Mainly, there are two types of lists: static and dynamic lists. Stack, queue and Random array, are all different forms of lists. According to its needs, an application may need to use a static queue or a dynamic queue, a static stack or a dynamic stack and so on. Figure (2-1) and Example 1 below illustrate the traditional way which is followed in such cases.

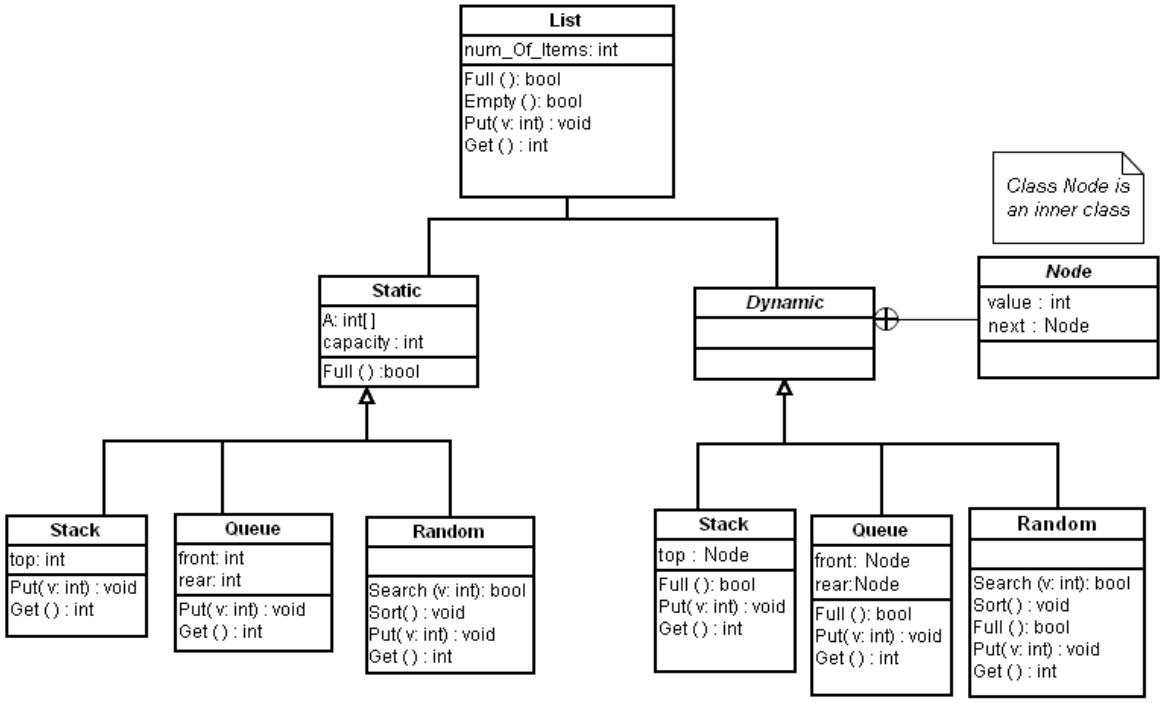


Figure (2-1): Hierarchy of List example, in a traditional approach -using UML notations.

```
abstract class List
{
 protected num_Of_Items;

 abstract void Put(int v);
 abstract int Get( );
 abstract bool Full ( );
 public bool Empty ( )
 {...}
...
}
```

```
abstract class Static extends
List
  {
   protected int[size] A;
   protected int capacity;
   public bool Full( )
   {...}
   ...
  }
```

```
Public class Stack extends
Static
{
  private int top;

  public Stack( ) {...}
  public void Put(int v) {...}
  public int Get( ) {...}
   ...
}
```

```
public class Queue extends
Static
{
 private int front;
 private int rear;

 public Queue( ) {...}
 public void Put(int v) {...}
 public int Get( ) {...}
 ...
}
```

```
public class Random extends
Static
{
  public Random( ){ ...}
  public void Put(int v) {...}
  public int Get( ) {...}
  public void Sort ( ) {...}
  public bool Search (int v) {...}
 ...

}
```

Example 1: Java code of Static lists, using a traditional approach

Some applications may use an object of type static stack; afterwards it would possibly want to change the type of the object into a dynamic queue. To capture this need we use a *genome* class (Example 2), which includes the definition of all potential characteristics (structures, operations and behaviors). To capture the differences between dynamic and static lists, two different *variants (genotypes)* are used, which are Static_List (Example 3) and Dynamic_List. Objects of previous variations are called *phenotypes* (Example 4). Phenotypes are similar to real entities.

In addition to structures, operations and behaviors, the genome definition contains additional information called *genetic evolution relations* (GER). Some of them are found in Example 2 (AreDominant, AreImplied, AreExclusive, etc.).

```
Class List (Type T, int Size)

  // Potential characteristics
 Instance Data
   int age;
   Struct Node { T value; Node *next;}
   ListStr=Alt{int Front;T A[Size];int Last;}
   //Static
              {Node *Front; Node *Last ;}
   //Dynamic
           EndAlt

 Instance Methods
  PutRandom-A(T val):Alt {//in static list};
                         {//in dynamic list};
                       EndAlt
  PutAtEnd-L(T val):Alt {//of static list};
                        {//of dynamic list};
                      EndAlt

  T GetRandom-A( ): Alt {//of static list};
                        {//of dynamic list};
                     EndAlt
  T GetFromBeg-L( ): Alt{//of static list};
                        {//of dynamic list};
                      EndAlt
  T GetFromEnd-L(T val:Alt{//of static list};
                         {//of dynamic list};
                       EndAlt
  bool Search(T val):Alt{//in static list};
                        {//in dynamic list};
                     EndAlt
  void Sort( ):Alt{//of dynamic list};
                  {//of Static list};
               EndAlt
  bool Empty ( ): Alt{//of dynamic list};
                     {//of static list};
                   EndAlt
  bool Full( ):Alt {//of dynamic list};
                   {//of static list};
                EndAlt
// Control process
Behaviors
 Alt
   { //Static
   Q-Beh0 {…}
   Q-Beh1 {…}
   Q-Beh2 {…}
   S-Beh0 {…}
   R-Beh0 {…}
   }
   { //Dynamic
   Dy-Q-Beh1 {…}
   Dy-B-Beh0 {…}
   Dy- R-Beh0 {…}
   }
 EndAlt
```

```
Genetic Evolution Relations
    Dominant
    Front, Last, A /*Default priority. In case
    of conflicts the priority is given,
    respectively, to Front, last, and A.*/

    Enable    A; // A must be always enabled

    Imply
    Enable    Front →Enable (GetFromBeg-L and
    Empty and Full); // Queue

    Disable   Front   →Disable   GetFromBeg-L;
    // Rules of elements coherence

    Enable   Last →Enable (PutAtEnd-L and Empty
    and Full);    // Stack

    Disable    Last    →Disable    PutAtEnd-L;
    // Rules of elements coherence

    Disable (PutAtEnd-L and GetFromBeg-L and
    GetFromEnd-L)→Disable (Front and Last);
    // Rules of elements coherence

    Disable   Last   →Disable   (Front   and
    GetFromEnd-L);
    // Rules of elements coherence

    Enable GetFromBeg-L→Enable (PutAtEnd-L and
    Empty and Full); // queue

    Enable GetFromEnd-L→Enable (PutAtEnd-L and
    Empty and Full); // stack

    Exclude

    Disable (Front and Last)→Enable (PutRandom-
    A and GetRandom-A and Search and Sort);
    // simple array

    Enable GetFromBeg-L →Disable GetFromEnd-L;
    // Rules of elements coherence

    Enable GetFromEnd-L →Disable GetFromBeg-L;
    // Rules of elements coherence

    Enable       (Front and Last)  →Disable
    (PutRandom-A and GetRandom-A and Search and
    Sort); // Rules of elements coherence

    …

End List
```

Example 2: A Genome for List objects modelled by an extended class concept. Static and Dynamic lists are genotypes (variants) of this genome. Real objects (instances) of Random array, Queue, and Stack will be phenotype of corresponding genotypes.

```
Genotype Static_List

 // Potential characteristics

Instance Data //Static
 int age;
 int Front; int Last; T A[Size];

Instance Methods
 PutRandom-A(T val)//in static list
 PutAtEnd-L(T val)//of static list
 T GetRandom( )// of static list
 T GetFromBeg-L( )//of static list
 T GetFromEnd-L( ) //of static list
 bool Search(T val)//in static list
 void Sort( )//of static list
 bool Empty ( ) //of static list
 bool Full ( )//of static list

//Control process
Behaviors
 Q-Beh0 {...}
 Q-Beh1 {...}
 Q-Beh2 {...}
 S-Beh0 {...}
 R-Beh0 {...}

Genetic Evolution Relations
 {...}

End Static-List
```

Example 3: A genotype (variant) Static_List of the genome class List (Example 2).

```
// Static Queue State

Instance Data //Static
  int age;
  int Front;
  T A[Size];
  int Last;

Instance Methods //Static
  PutAtEnd-L (T val)
  T GetFromBeg-L ( )
  bool Empty ( )
  bool Full( )


Behaviors
  Q-Beh0{...}
  Q-Beh1{...}
  Q-Beh2{...}


Genetic Evolution Relations
 {...}
```

Example 4: An object (phenotype) Queue of variation Static_List (Example 3)

CHAPTER THREE

LITERATURE REVIEW

In this chapter, we present some current approaches in object evolution, followed by some fault tolerance approaches. We finalize this chapter with a conclusion about common weaknesses which justify our presented work.

3.1 Object Evolution Approaches

There have been different approaches to deal with object evolution. In this section we consider some approaches that are most related to our approach.

Object re-classification (D.Ancona et al, 2007)**.** Object re-classification can be defined as changing the class membership of an object while retaining its identity, at run time. This work tried to come up with new language features; this was achieved through a language called Fickle and its extensions FickleII and FickleIII. The suggested language features allow objects to change class membership dynamically.

State classes are possible targets of re-classifications; they represent object's possible states. Root classes are the super-classes of such state classes and declare all the members common to them. Fickle provides some annotations before methods bodies, called *effects*. Effects list the root classes of all objects that may be reclassified by invocation of this method. Methods with empty effects may not cause any reclassification. Our case study could be represented using this approach as illustrated in Figure (2-1). Example 5 illustrates how our case study can be applied using Fickle language. Evolution between dynamic list sub-classes could be done in a similar way.

Object evolution (T.Cohen and J.Gily, 2009). Object evolution allows objects to change their classes at runtime. It has been considered as a restriction of dynamic object re-classification. Object evolution, allows an object to gain, but never lose capabilities.

A new function was introduced in this work, which is called *Evolver function.* It is used as a complementary mechanism to constructors. It contains the additional initialization code that separates an object of one class from an object of another. Like constructors; evolvers can accept parameters, indicating that an object cannot be evolved into a new class without some additional required information.

```
abstract class List
{
  protected int num_Of_Items;
  abstract void  put (int v ) { };
  abstract  int get ( ) { Static};
  abstract  bool Full ( ) { } ;
  bool Empty ( )
  { retrun (num_Of_Items==0);}
  ….   // the rest of the class
}
```

```
abstract root class Static extends
List
{
  protected int[ ] A;
  protected in capacity;

  public bool Full( )
  { retrun (num_Of_Items==capacity);}
  …   // the rest of the class
}
```

```
State class Stack extends Static
{
 int top;
 Stack(int c)
 {
   top = -1;
   num_Of_Item=0;
   capacity=c;
 }
 void put( int v)
 {
   top++;
   A[top] = v;
   num_Of_Item++;
 }
 int get ( ) {Static}
 {
   int val = A[top];
   top--;
   this!!Queue ; this.front= 0;
   this.rear =  top ;
  // Stack will turn into Queue [1]
   num_Of_Items--;
   return val;
 }
 …. // the rest of the class
}
```

```
State class Queue extends Static
{
 int front;
 int rear;
 Queue(int c)
 {
    front = 0; rear = -1;
    num_Of_Item=0;
    capacity= c;
  }
  void put( int v)
  {
    rear++;
    A[rear] = v;
    num_Of_Item++;
  }
  int get ( ) {Static}
  {
    int val = A[front];
    front++;
    this!!Stack ; this.top= front;
   // Queue will turn into Stack [2]
    num_Of_Item--;
    return val;
  }
… // the rest of the class
}
```

Example 5: Our case study using Fickle language. In [1], an object of type Stack will evolve into a Queue. In [2], an object of type Queue will evolve into a Stack.

An object evolution operation replaces, at runtime, the type of an object with the type of a selected subclass. Our case study could be represented using this approach as explained in Figure (2-1). Example 6 illustrates how our case study is applied using this approach. Evolution between dynamic list sub-classes could be done in a similar way.

```
abstract class List
{
  protected int num_Of_Items;
  abstract  void  put (int v ) ;
  abstract  int get ( ) ;
  abstract bool Full ( ) ;
  public bool Empty ( )
  {return(num_Of_Items==0);}
  … // the rest of the class
}
```

```
abstract class Static extends List
{
  protected int[ ] A;
  protected int capacity;
  public bool full ( )
  {return num_Of_items==capacity;)}
   …// the rest of the class
}
```

```
class Stack extends Static
{
 int top;
 Stack( int c)
 {
   top = -1;
   num_Of_Item=0;
   capacity= c;
 }
 void put( int v)
 {
   top++;
   A[top] = v;
  }
 int get ( )
 {
   int val = A[top];
   top--;
   this → Queue( 0)  ;
   // Stack will turn into Queue [2]
   return val;
 }
 … // the rest of the class
}
```

```
class Queue extends Static
{
  int front;
  int rear;
  Queue(int c )
  {
    front = 0; rear = -1;
    num_Of_Items=0;
    capacity = c;
  }
  → Queue( int front )// Evolver [1]
  {
  this.front=front;
  }
  void put( int v)
  {
    rear++;
    A[rear] = v;
  }
  int get ( )
  {
    int val = A[front];
    front++;
    return val;
  }
… // the rest of the class
}
```

Example 6: Our case study using Object Evolution approach. In [1], an Evolver function is defined, with additional information (front). In [2], an object in Stack state will evolve into a Queue using the Evolver function.

Class evolution (E.Johnsen et al, 2009). Supporting an arbitrary set of changes to classes which may have large number of persistent objects. At runtime, class redefinitions gradually upgrade existing instances of a class and of its subclasses. An upgrade may depend on previous upgrades of other classes. This work came up with a modeling language which supports the runtime evolution of distributed object-oriented systems. This language has dynamic class operations, which can introduce new functionality and interfaces for classes, change data structures and implementations for existing functionality, and remove legacy code. Our case study could be represented using this approach as explained in Figure (3-1) and Example 7.

Schema evolution (M.Piccioni et al, 2011). Schema evolution allows old objects to fit into new classes. In this work, refactoring units are used to modify the attributes. Refactoring is a function modifying at most one attribute; it allows an easy representation of the static transformations of a class. In addition to refactoring units, object transformers are used to create an instance of the new class from a serialized instance of the old class. The generation of object transformers from a class transformation can be expressed as transformation functions. The representation of our case study using this approach is illustrated in Figure (3-1) and Example 7.

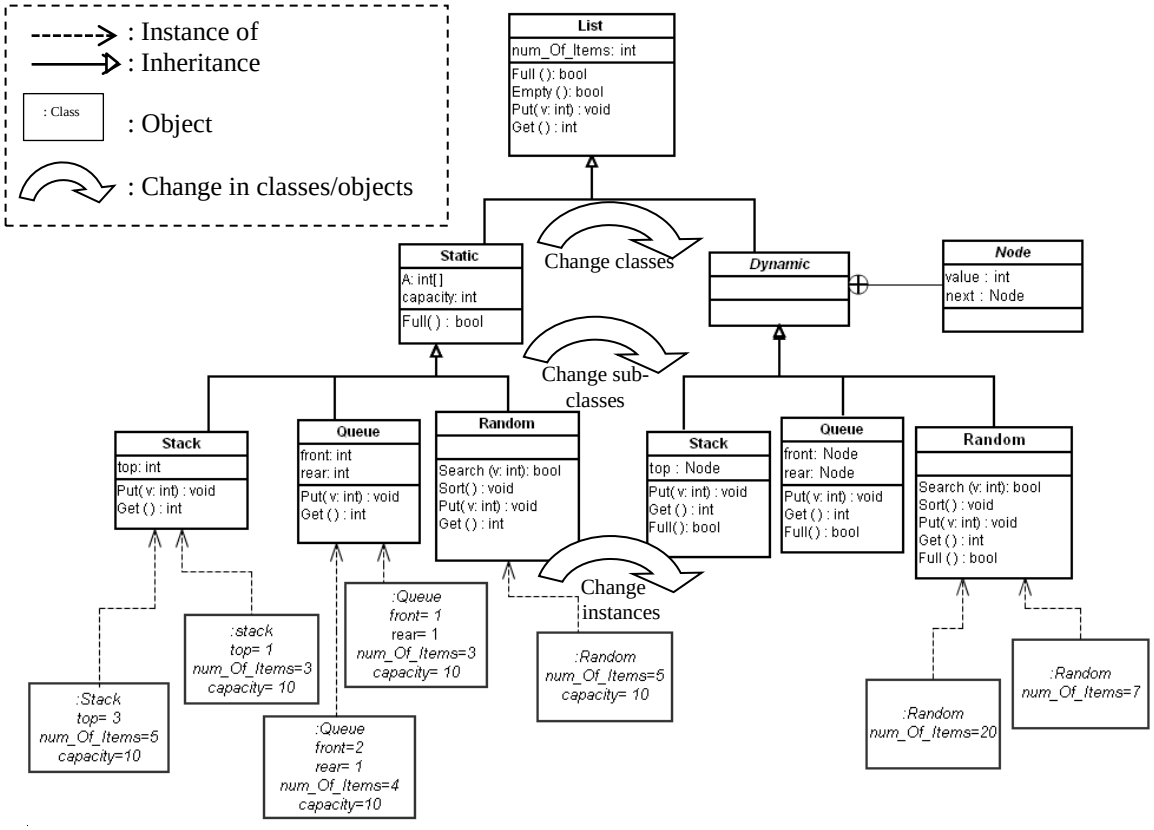
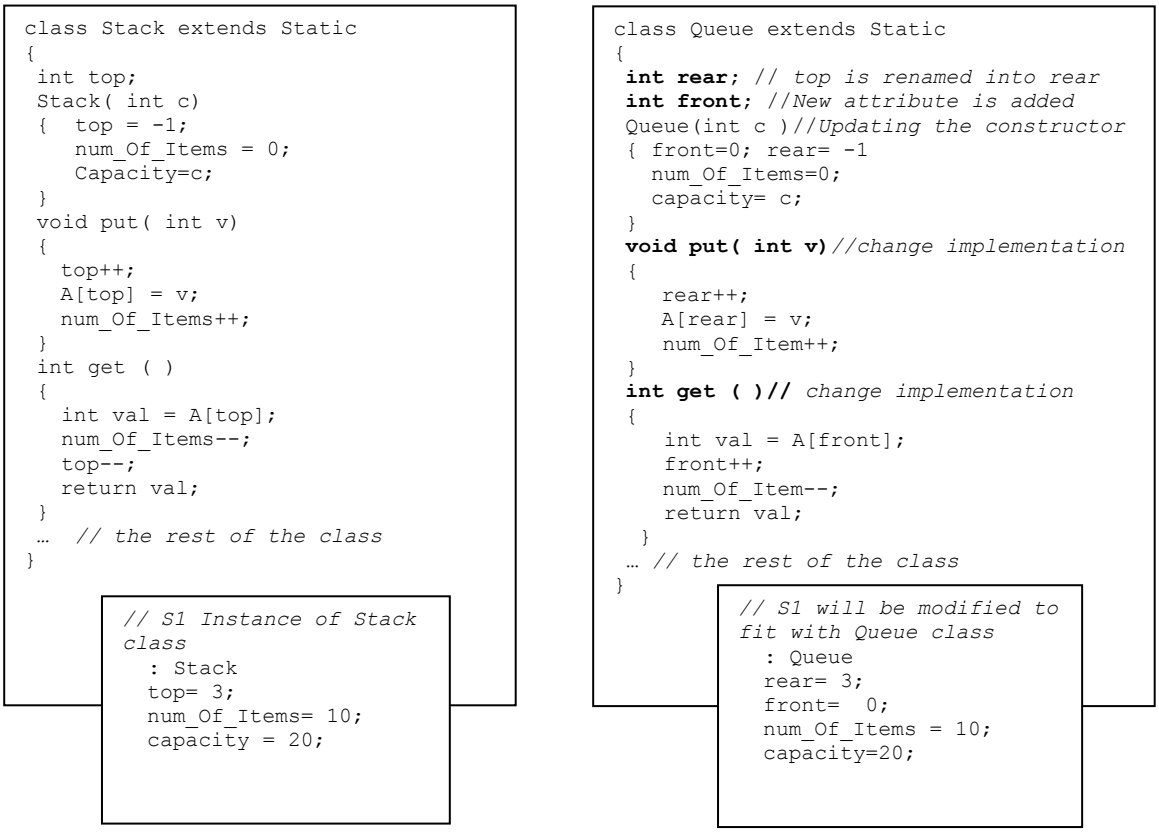


Figure (3-1): Classes and objects hierarchy following class evolution and schema evolution approaches.

Roles. There have been different proposals for extending traditional object model with Role mechanisms. Roles define extra properties which are added to objects. During its lifetime, an object may adopt and abandon roles (D.Stein et al, 2005).

Usually, the same object may be perceived differently depending on other objects it is collaborating with. Role models identify roles as types and describe the network of roles required for a specific collaboration to happen. As a player of collaboration, a role defines the set of extrinsic properties and behavior necessary to realize its participating collaborations. Roles are modeled as classes and represented in class diagrams. Methods and

attributes concerning the specific collaboration context can be included in this class diagram (A. Caetano et al, 2005).



```
class Stack extends Static
{
 int top;
 Stack( int c)
 {  top = -1;
    num_Of_Items = 0;
    Capacity=c;
 }
 void put( int v)
 {
   top++;
   A[top] = v;
   num_Of_Items++;
 }
 int get ( )
 {
   int val = A[top];
   num_Of_Items--;
   top--;
   return val;
 }
 …  // the rest of the class
}
```

```
// S1 Instance of Stack
class
   : Stack
   top= 3;
   num_Of_Items= 10;
   capacity = 20;
```

```
class Queue extends Static
{
 int rear; // top is renamed into rear
 int front; //New attribute is added
 Queue(int c )//Updating the constructor
 { front=0; rear= -1
   num_Of_Items=0;
   capacity= c;
 }
 void put( int v)//change implementation
 {
    rear++;
    A[rear] = v;
    num_Of_Item++;
 }
 int get ( )// change implementation
 {
    int val = A[front];
    front++;
    num_Of_Item--;
    return val;
  }
 … // the rest of the class
}
```

```
// S1 will be modified to
fit with Queue class
   : Queue
   rear= 3;
   front=  0;
   num_Of_Items = 10;
   capacity=20;
```

Example 7: Updating Stack class: Adding a new attribute front, renaming top into rear, changing the implementation of put and get; resulting in class Queue. Object S1 of type Stack is modified to fit with the new class Queue.

Dynamic role concept as mentioned in (A.Jodloowski et al, 2004) assumes that every real or abstract entity during its life can acquire and lose many roles without changing its identity. Roles appear during the life of a given object, they can exist simultaneously, and they can disappear at any moment. Roles are treated as a special kind of objects. Also, as in the case of regular objects, classes describing roles can be specialized. The approach followed in (A.Jodloowski et al, 2004) assumes that an object can contain many sub-objects called roles which can be added/removed at run time. A role has its own attributes and behavior. It dynamically "imports" attributes (values) and behavior from its super-roles, in particular, from its parent object. Our case study could be represented using this approach as illustrated in Figure (3-2).

Some works have been conducted in the area of Object Role Modeling (ORM). ORM includes graphical and textual language for modeling and querying information at the

conceptual level as well as procedures for designing conceptual models, transforming between different conceptual representations (T.Halpin, 2010). However, the term *object* is used in ORM, in a way that is different from the way it is used in an object-oriented model (OOM). In OOM objects are dynamic entities, not static ones, i.e. objects have operations in addition to structures.

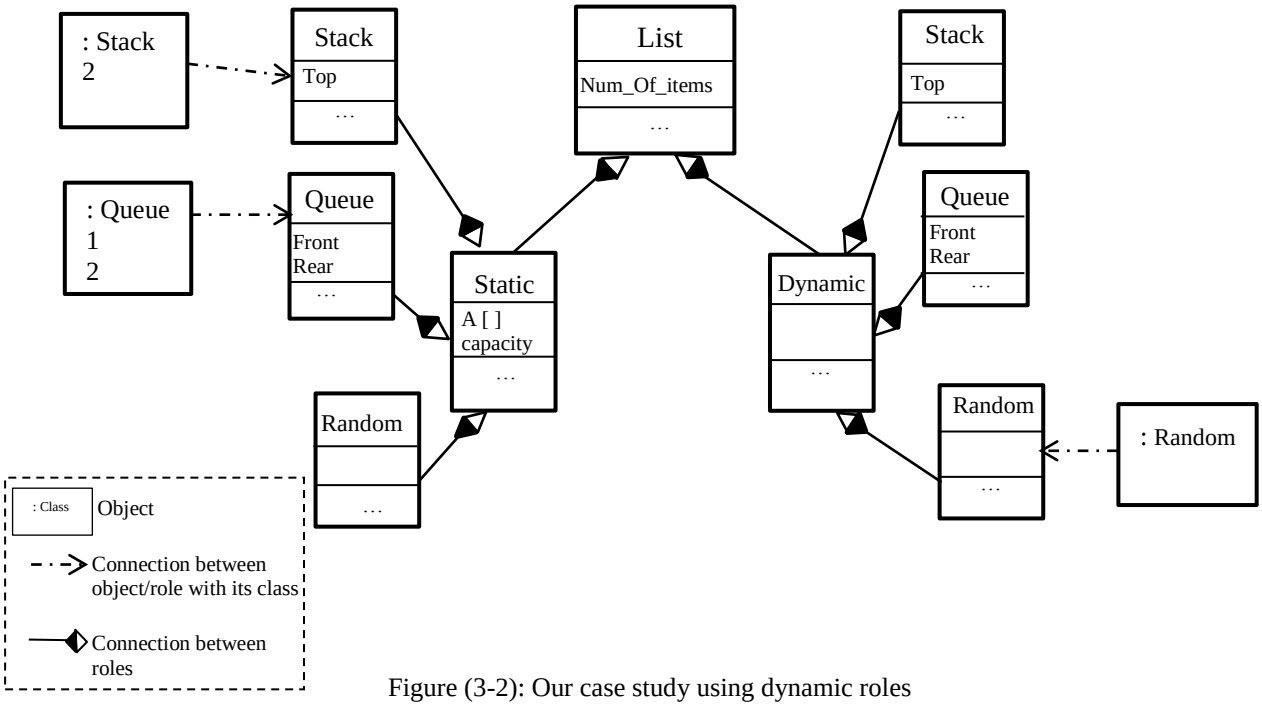


Figure (3-2): Our case study using dynamic roles

Our case study could be represented using this approach as it follows in Figure (3-3):

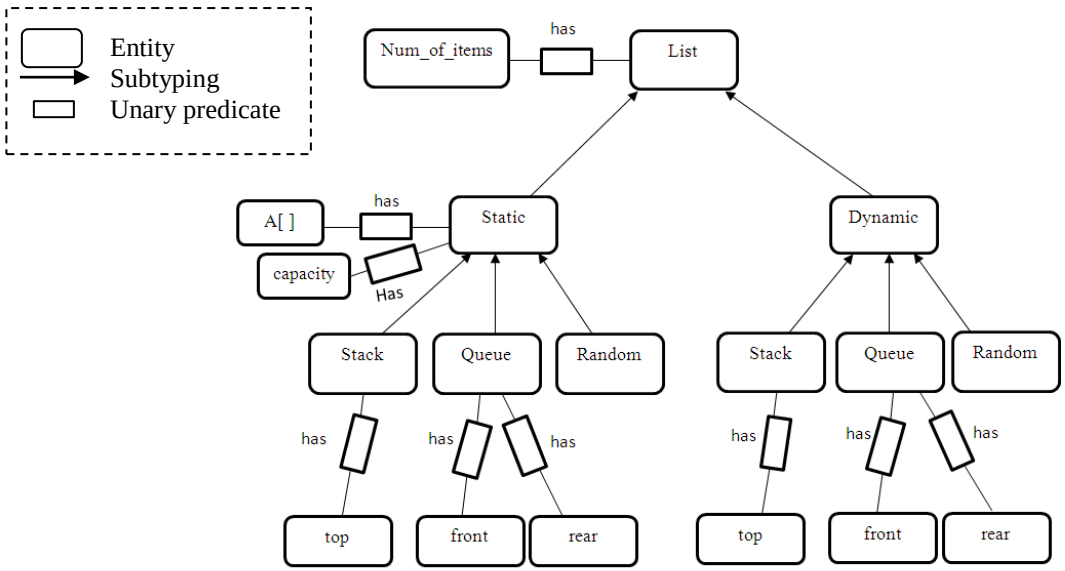


Figure (3-3): our case study using ORM

Bio-inspired approach. The work of (D.Meslati and S.Ghoul, 2005) presents a model for the change process of software systems based on biological concepts. Their approach proposes a model where anticipated and unanticipated changes are modeled by a collection of fine grained instructions called genes.

The software system consists of structural, behavioral and a third dimension called *ontogenesis*. All changes undergone by a software system are considered as its *ontogenetic* dimension. In (S.Ghoul, 2010), a bio-inspired integrated model is presented, and the space of bio-inspired systems is partitioned into three axes: phylogeny, ontogeny, and epigenesis (Figure 3-4). The *ontogen* [illegible] involves the *growth* or *construction* of a phenotype (single individual) from its own genotype (its genetic material), essentially without environmental (peristase) interactions.

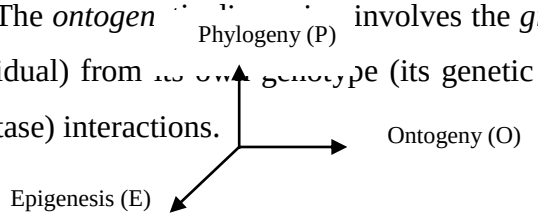


Figure (3-4): The POE model. Partitioning the space of bio-inspired systems along three axes: Phylogeny, Ontogeny, and Epigenesis concepts

The *genome* (D.Hammodeh, 2012), (S.Ghoul, 2010) of a species includes the definition of all its possible characteristics (organic, functional, and behavioral) along with the information controlling their coherence. A *genotype* (D.Hammodeh, 2012), (S.Ghoul, 2010) is a coherent partition of genome characteristics, obtained by selective inheritance (D.Hammodeh, 2012). This coherence deals with selecting no contradictory characteristics describing exactly a partition of objects (phenotypes) of the associated species. A *phenotype* (S.Ghoul, 2010) is an instance of a given genotype. In the artificial world, several phenotypes may be instances of the same genotype. In real world, from each genotype only one instance might be developed. The physical development and phenotype of organisms can be thought of as a product of genes interacting with each other and with the environment.

The genome contains additional information called *genetic evolution relations*. These relations are composed of control genes which enforce and control the coherence of genome functions by establishing and managing dependencies relations between its elements (S.Ghoul, 2010). The following are some common identified control genes:

AreExclusive genes*:* These genes identify the characteristics that are exclusive. A characteristic excludes another if they are alternatives (versions) or they are incompatible.

AreExclusive gene: ***If*** *Enable organ definition/function gene* ***Then***

Disable organ definition/function gene

AreImplied genes*:* A characteristic implies another if its presence in a phenotype implies the presence of the other. *AreImplied genes* ensure the implication between enabled and disabled genes.

AreImplied gene: ***If*** *Enable/Disable organ definition /function gene1 Then*

Enable/Disable organ definition/function gene2

3.2 Fault Tolerance Approaches

In this section we consider some fault tolerance approaches, on which our fault tolerance technique is based.

In software engineering, critical software systems must be fault tolerant. This is required when there are high availability requirements or when system failure costs are very high. Fault tolerance means that the system can continue in operation in spite of software failure. Even if the system has been proved to conform to its specification, it must also be fault tolerant as there may be specification errors or the validation may be incorrect (I.Sommerville, 2011). Sommerville defines a fault tolerance process as it follows:

Fault detection: The system must detect that a fault (an incorrect system state) has occurred.

Damage assessment: The parts of the system state affected by the fault must be detected.

Fault recovery: The system must restore its state to a known safe state.

Fault repair: The system may be modified to prevent recurrence of the fault. As many software faults are transitory, this is often unnecessary.

In biology (B.Alberts et al, 2007), although the DNA is a highly stable material, as required for the storage of genetic information, it is a complex organic molecule that is susceptible to accidents, even under normal cell conditions. Spontaneous changes may lead to mutations if left unrepaired. The double-helical structure of DNA is ideally suited for repair because it carries two separate copies of all the genetic information-one in each of its two strands. Thus, when one strand is damaged, the complementary strand retains an intact

copy of the same information, and this copy is generally used to restore the correct nucleotide sequences to the damaged strand.

3.3 Literature Review Conclusion

There have been different approaches to handle objects evolution. The followed approaches did not conform to real entities evolution; the gap between their concepts and real world concepts was large. This has led to some insufficiencies; we had to deal with large number of classes, unnecessary hierarchies; causing the design and the implementation to become more complex. An object of real world may evolve during its lifetime without changing its membership, and without the need to update its original class. It has become clear that we need to apply biological concepts on objects evolution to come up with good and acceptable results. The two works (D.Meslati and S.Ghoul, 2005) and (S.Ghoul, 2010), provided a general background in both, bio-inspired modeling and development. Our work builds on these two works.

CHAPTER FOUR

A GENETICS-BASED METHODOLOGY TO OBJECT EVOLUTION

Artificial objects, just like biological entities, evolve organically and behaviorally. The organic evolution deals with the structures and functions of objects, while the behavioral evolution deals with the behavior of objects. Genetics-based evolution is predefined and pre-planned in a genetic evolution process that defines the object lifecycle. This evolution may deal with objects or with the evolution process itself (this kind of evolution is out of scope of this work).

In the following, we introduce some genetics-based evolution concepts based on the needs stated in (D.Meslati and S.Ghoul, 2005) and (S.Ghoul, 2010). An object may evolve inside one genotype, and/or between different genotypes. This evolution can be organically (structures and functions) and behaviorally.

4.1 Genetics-Based Evolution Methodology

The proposed genetics-based evolution methodology is illustrated in Figure (4-1). It is based on genetic evolution programs specification, genetic evolution processes specification, genetic evolution relations specification and genetic evolution control. Firstly, the genetic evolution information that exists within the selected genotype is used as an input to the genetic evolution control, in order to create a phenotype (instance of a genotype). This process is only executed once; at the creation of phenotypes (age=0). The genetic evolution information within the created phenotype is used as an input to the genetic evolution control, resulting in an evolved phenotype (age > 0). This process is executed each time a phenotype evolves, till it reaches the termination age x, where 0 < age <x.

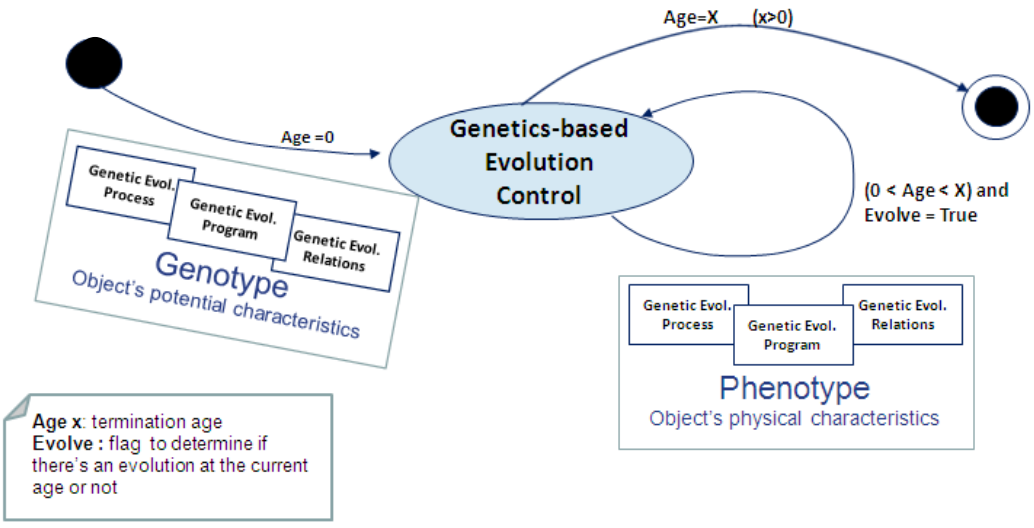


Figure (4-1): Genetics-based evolution methodology

If the genetic evolution control is executed without failures, then the resulted evolution is guaranteed to be complete and accurate.

4.2 Genetics-Based Evolution Concepts

In this section, we will introduce some key concepts of genetics-based methodology to object evolution, which will be used throughout this study.

- **Genetic Evolution Program (GEProg)**

  A genetic evolution program is a program that specifies what structures and operations are needed for a certain evolution (Example 8). This program is specified as it follows:

```
GEProg <Id>
{
    Enable (<Structure>,)* | (<Function>,)*;
    Disable (<Structure>,)*| (<Function>,)*;
}
```

  Where:

  "|" means OR, and *"*"* means repeated once or more.

  Enable is a predefined operation that allows an object of a genotype to hold enumerated structures or functions.

  Disable is a predefined operation that allows an object to lose enumerated structures or functions. At the initial state, all the structures/functions of the genotype are disabled. The disabled structures/functions are inactive. Inactive elements cannot be used until they are enabled.

- **Genetic Evolution Relations (GER)**

  In addition to the genetic relations that are mentioned in Chapter One (AreImplied, AreExculsive, etc.), we present a new genetic relation to insure the coherency of objects.

  For some objects, the existence/non-existence of an element X depends on the existence of element Y and non-existence of element Z. The new defined rule will be helpful in similar cases.

  Compound genes: These genes are considered as a combination of *AreImplied* genes and *AreExclusive* genes:

  If Enable (organ definition/function gene) and Disable (organ definition/function gene)
  Then Enable/Disable (organ definition/function gene).

  Example: *Enable Last and Disable Front* → *Enable* GetFromEnd-L; *//stack*

- **Genetic Evolution Process (GEProc)**

A genetic evolution process is a process that defines the lifecycle of an object, by determining what evolution is needed at what time (Figure 4-4). This process is specified as it follows:

GEProc <id>
{
((age = A_i): $GEProg_i$, $Behavior_i$)*;
}

Where A_i is an age milestone, $GEProg_i$ is the corresponding genetic evolution program to achieve, $Behavior_i$ is the evolved-to behavior (Figure 4-2). And "*" means repeated once or more.

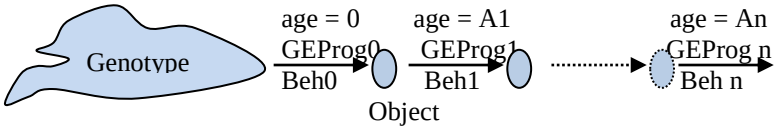


Figure (4-2): Genetic evolution process

The *age* is a hidden attribute of an object inherited automatically from its genotype. It is initialized to zero at its creation. Each time an object is used this attribute is updated by computing the difference between the actual date and the date of its generation. The unit of *age* can be measured by year, month, day, minute, second, etc.

- **Genetics-Based Evolution Control (GEC)**

The interpretation of an evolution program is mainly supported by the genetic evolution relations which ensure the coherence of the evolution process. The inter/intra relation coherence is ensured at its definition or update.

The genetics-based evolution control enforces the following rules on its associated genotype (Example 3) and phenotype (Example 4).

- ***Initial state: Elements and coherence***

R1. Each object holds, from its genotype, an initial set of structures and functions defined by its evolution program at its creation. All these structures/functions are disabled.

R2. Let Enabled_List be the list of the structures and functions to be enabled.

Enabled_List ← structures and functions to be enabled (imposed by an Enable clause in the GEProg);

R3. Let Disabled_List be the list of the structures and functions to be disabled.

Disabled_List ← structures and functions to be disabled (imposed by a Disable clause in the GEProg);

R4. The coherence of Enabled_List and Disabled_List is checked: Enabled_List ∩ Disabled_List = Ø. Each element of Enabled_List does not imply directly or indirectly an element of Disabled_List.

- ***Enable/Disable List processing by scanning genetic evolution relations***

  R5. The processing of enable/ disable list is obtained, by scanning the genetic evolution relations as it follows:

  - For each element in the Disabled_ List, not processed yet: (1) Disable the element. (2) Find disabled structures and disabled functions associated with it. Add them to Disabled_List. (3) Find enabled structures and functions associated with the element. Add them to Enabled_List.
  - For each element in the Enabled_List, not processed yet: (1) Enable the element. (2) Find enabled structures and enabled functions associated with it. Add them to Enabled_List. (3) Find disabled structures and functions associated with the element. Add them to Disabled_List.
  - Check for coherence when adding new elements to the lists.

- ***Loop on Enable list and Disable list processing***

  R6. Repeat R4 and R5 until all their elements are processed.

- ***Final state***

  R7. Coherence errors cause failure of the GEC. If this process succeeds, Enabled_List will contain the structures and functions which are enabled, Disabled_List will contain the structures and functions which are disabled.

Figure (4-3) shows the GEC algorithm, written in pseudo code.

4.3 Genetics-Based Evolution Scope

In the genetics-based methodology to object evolution, an object can evolve inside its genotype, or between different genotypes (from one genotype to another).

- **Evolution Inside a Genotype**

Inside a genotype, an object may evolve organically by holding/losing structures and functions of its actual genotype and behaviorally by holding/losing behaviors. Just like natural evolution, this evolution is pre-planned in a genetic evolution process, inherent to a genotype, defining the genotype objects lifecycle.

```
FUNCTION GEC( )
BEGIN
 Read GEProg( );
 Add to_enable structures,functions to Enabled_List;
 //to_enable:  elements preceded by Enable clause

 Add to_disable structures,functions to Disabled_List;
 //to_disable: elements preceded by Disable clause

 coh_status = Check_coherence( );
 //call Check_coherence function

 IF Coh_status= success THEN Process_elements( );
END  FUNCTION
```

```
FUNCTION Check_coherence( )
BEGIN
  IF element E ∈ (Enabled_List ∩ Disabled_List)
  THEN
     return fail;
  ELSE
     return success;
  END IF
END FUNCTION
```

```
FUNCTION Scan_GER(string element )
BEGIN

//imply
IF element ∈ imply_List THEN
  value = evaluate Left hand side Expression;
  //True or False
  IF value = True And element ∈ Enabled_List THEN
        Add associated elements to Enabled_List;
  IF value = True And element ∈ Disabled_List THEN
        Add associated elements to Disabled_List;
//exclude
IF element ∈ exclude_List THEN
  value = evaluate Left hand side Expression;
  //True or False
  IF value=True and element ∈ Enabled_List  THEN
      Add associated elements to Disabled_List;
  IF value= True and element ∈ Disabled_List  THEN
      Add associated elements to Enabled_List;

//compound
IF element ∈  compound_List THEN
    value = evaluate Left hand side Expression;
    //True or False
    IF value = true THEN
        enable/disable associated element
        //according to the defined relation

END FUNCTION
```

```
FUNCTION Process_elements( )
BEGIN
  FOR each element E ∈ Enabled_List  And E not
processed
    BEGIN
    Enable E;
    Scan_GER( E );
    coh_status= Check_coherence( );

    IF coh_status = success Then
        Mark E as processed;
    ELSE
        Return fail;
    END IF
  END FOR

  FOR each element E ∈ Disabled_List And E not
processed
    BEGIN
    Disable E;
    Scan_GER( E );
    coh_status= Check_coherence( );

    IF coh_status =  success THEN
        Mark E as processed;
    ELSE
        Return fail;
    END IF
  END FOR
END FUNCTION
```

Figure (4-3): GEC algorithm, in pseudo code.

At its creation, each object holds its own lifecycle that determines the needed evolution to be achieved genetically and automatically. Once an object is created, it holds an initial genetic information subset of its genotype; defined explicitly and implicitly by its initial genetic evolution program (structures, and functions). Example 8 shows two evolution programs; StQueue and StStack.

Each object has an age defining milestones through its lifecycle. At each milestone, the object evolves automatically from one state to another. Naturally, the environment may influence this evolution at any time during the object lifecycle. This influence, carried out genetically, is out of scope of this work. While the evolution by environment influence affects specific objects, the genetic evolution relates to all objects of the associated genotype. Figure (4-4) shows a genetic evolution process *Static*, associated to the genotype *Static_List* (Example 3); defined textually and graphically.

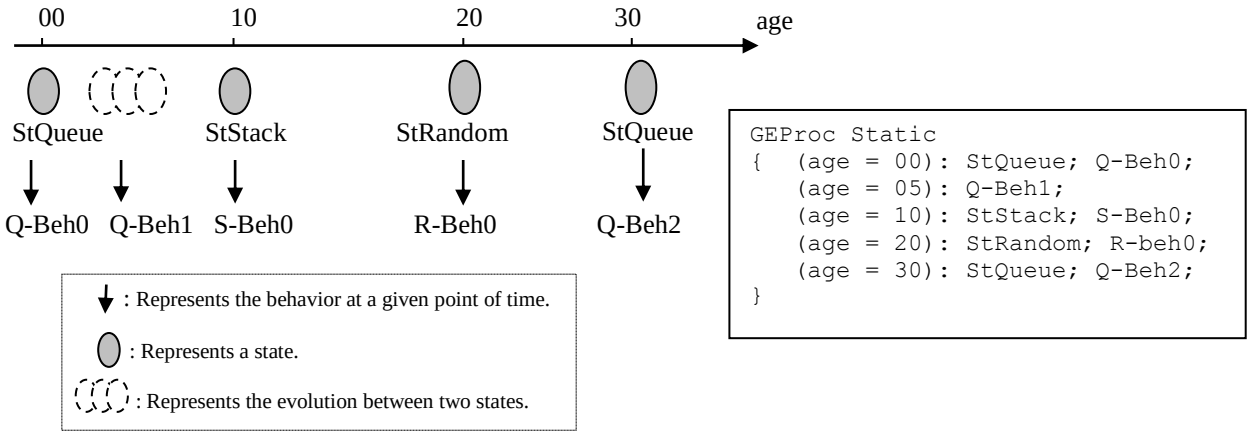


Figure (4-4): A genetic lifecycle (GEProc) of an object of Static_List genotype.

```
GEProg StQueue
{
  Enable front;
  Enable Last;
}
```

```
GEProg StStack
{
  Enable Last;
  Disable Front;
}
```

Example 8: Static Queue and Static Stack genetic evolution programs

The behavior of an object is associated to functions, so the behavioral evolution is a consequence of structural and functional evolution. A behavior of an object is an organization, in the time, of its state enabled functions. So, to each state is associated a behavior, and thus to the organic evolution is associated a behavior evolution (Figure 4-4). However, even at the same state, the object behavior may evolve in the time, separating it from next state. We define a phenotype behavior as it follows:

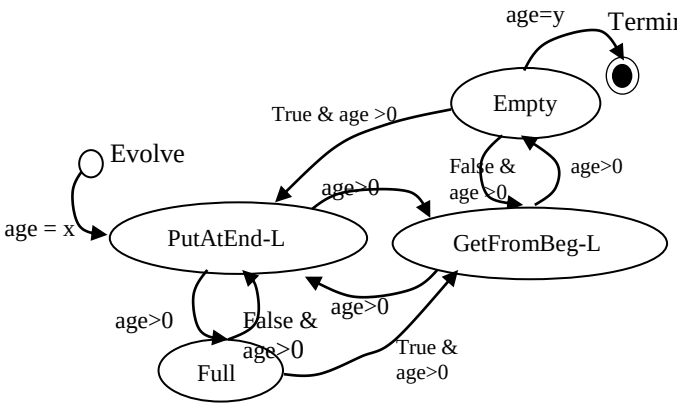
Behavior <id>
{
  <function> → ((<condition>)* <function>,)*;
}

Where:

→ : The right side functions will be executed after the left side functions.

( ) *: Repeated once or more.

The behavior *Q-Beh0* associated to a List in a *StQueue* state may be defined graphically and textually as it follows in Figure (4-5):




```
Behavior Q-Beh0
{
  Evolve →(age=x) PutAtEnd-L;
  PutAtEnd-L →(age>0) Full, (age>0)
  GetFromBeg-L;
  Full →(False)&(age>0) PutAtEnd-L, (True)&
  (age>0) GetFromBeg-L;
  GetFromBeg-L →(age >0) PutAtEnd-L, (age >0)
  Empty;
  Empty →(False)& (age >0)  GetFromBeg-L,
  (True) &(age>0) PutAtEnd-L, (age=y)
  Terminate;
}
```

Figure (4-5): Behavior Q-Beh0 of an object in StQueue state. x is the age when an object evolves into Q-beh0 state , y is the age when an object evolves into another state according to its GEProc.

The behavioral evolution process is enforced by the following rules:

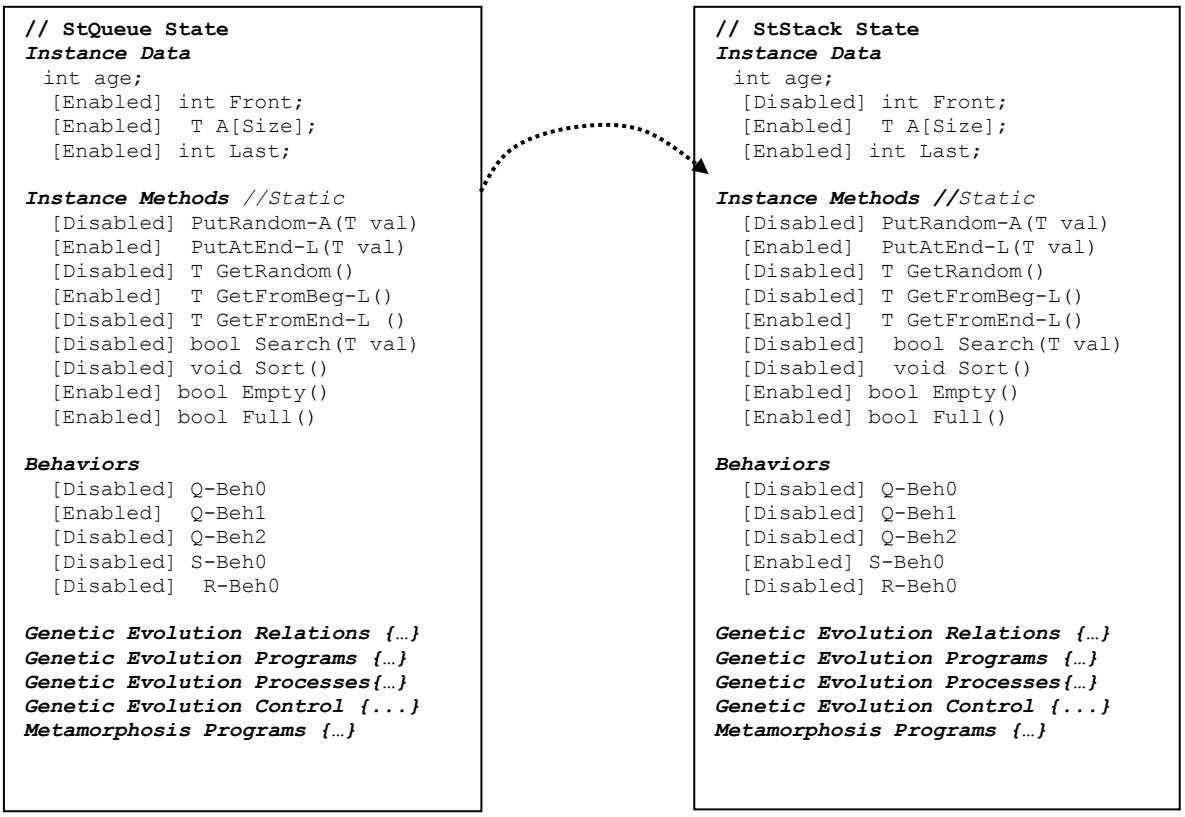
R1. All the involved functions must be enabled, at the associated evolution state, else the process stops.

R2. Labeled arrows are conditions on function outputs. Unless these conditions are met, the target functions will not be executed.

The following application program creates an object *List1* from the genome (class) List (Example 2), holding the features specified by the genotype *Static_List* (Example 3) and having the lifecycle defined by the GEProc *Static* (Figure 4-4)

```
{ …
 List Static_List List1= New (GEProc Static);
/* List is the genome class.
 Static_List is a genotype of List, defining potential characteristics to be held by objects of
 this genotype.
 List1 is an instance (phenotype), inheriting its characteristics from List according to
 Static_List requirements.
 GEProc Static is the GEProc defining the lifecycle of List1(Figure 4-4).
 List1 will have an initial state StQueue, defined by the GEProg StQueue (Example 8) and
 will behave according to the behavior Q-Beh0 (Figure 4-5)
 */
  . . . // List1 is used as a static Queue
}
```

At age = 10, *StStack* GEProg will be activated (as illustrated in Figure 4-4). It was designed for changing *StQueue* state to *StStack* state (Example 9).



Example 9: Phenotype evolution from StQueue to StStack (at age10).

- **Evolution Between Genotypes (Metamorphosis)**

Between genotypes an object may evolve by losing structures, functions, and behaviors of its actual genotype and holding structures, functions, and behaviors of another genotype. The evolution inside a genotype was studied previously, so this part deals with the evolution from one genotype to another, which we term by the *metamorphosis*, i.e. From static Queue list to dynamic Queue list, from dynamic Random list to static Random list, etc.

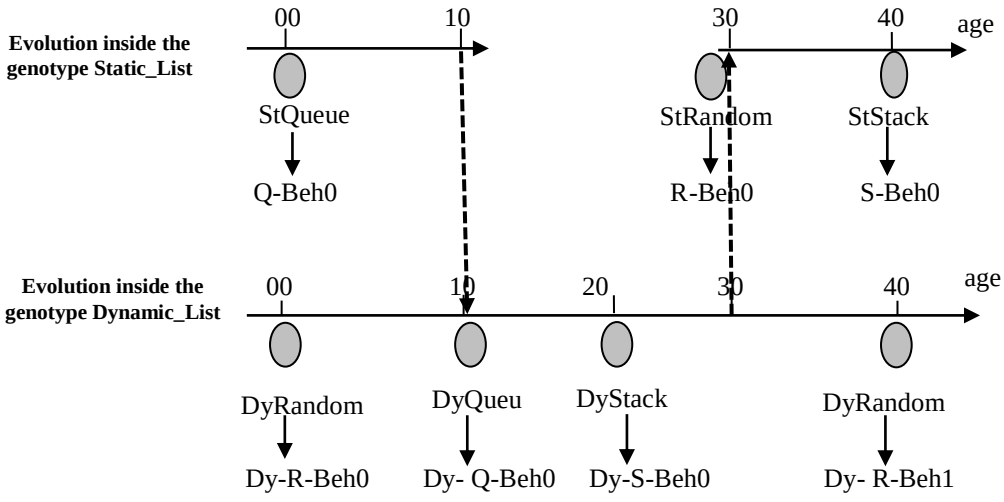
A *metamorphosis* is an evolution with change (increase, destruction) in structures, functions and behaviors, whereas the evolution is only in enabling or disabling already held (from the corresponding genotype) structures, functions and behaviors. Figure (4-6) shows a genetic evolution process of an object that evolves between two genotypes, *Static_List* and *Dynamic_List*, defined textually and graphically. The *Dynamic_List* genotype of class List (Example 2) is defined in Example 10.

```
Genotype Dynamic_List
// potential characteristics
Instance Data
  int age;
  Struct Node { T value; Node *next;}
  Node *Front; Node *Last;
Instance Methods
  PutRandom-A(T val)   //in dynamic list
  PutAtEnd-L(T val)    //of dynamic list
  T GetRandom( )       //of dynamic list
  T GetFromBeg-L( )    //of dynamic list
  T GetFromEnd-L( )    //of dynamic list
  bool Search(T val)   //in dynamic list
  void Sort( )         //of dynamic list
  bool Empty( )        //of dynamic list
  bool Full( )         //of dynamic list

//control process
Behaviors
  Dy-Q-Beh0 {…}
  Dy-S-Beh0  {…}
  Dy-R-Beh0 {…}

Genetic Evolution Relations
{…}
Genetic Evolution Programs
{…}
Genetic Evolution Processes
{…}
Genetic Evolution Control
{…}
Metamorphosis Programs
{…}
End Dynamic_List
```

Example 10: A Dynamic list genotype



: Represents the behavior at a given point of time.

: Represents a state.

: Represents a transition between different genotypes.

```
GEProc Static_Dynamic
{  (age= 00): StQueue, Q-Beh0;
   (age=10): StQueueToDyQueue, Dy-Q-Beh0;
   /*StQueueToDyQueue is a Metamorphosis program from
   StQueue to DyQueue*/
   (age=20): DyStack, Dy-S-Beh0;
   (age=30): DyStackToStRandom, R-Beh0;
   /* DyStackToStRandom is a Metamorphosis program from
   DyStack to StRandom */
   (age=40): StStack, S-Beh0;
}
```

Figure (4-6): A genetic lifecycle of an object evolving between Static_List and Dynamic_List genotypes, by metamorphosis programs.

The metamorphosis of an object O1, of a genotype G1 to a genotype G2, is a process which may change O1 completely (i.e. destruction of old structures and holding new ones). It operates like a conversion of O1 to a new object O2 of the genotype G2, with a maximum of information transition, such as identity, age, lifecycle, persistent state information, etc. A metamorphosis program is defined as it follows:

```
Metamorphosis_Program<Id>
{
   Metamorphose to genotype <genotype_Id>;
   At the Evolution State <GEProg_1> to the Evolution State <GEProg_2>;
   Information transition ensured by the function <Funct_Id>;
}
```

Where:

genotype_Id: Target genotype

GEProg_1: Current GEProg.

GEProg_2: Target GEProg.

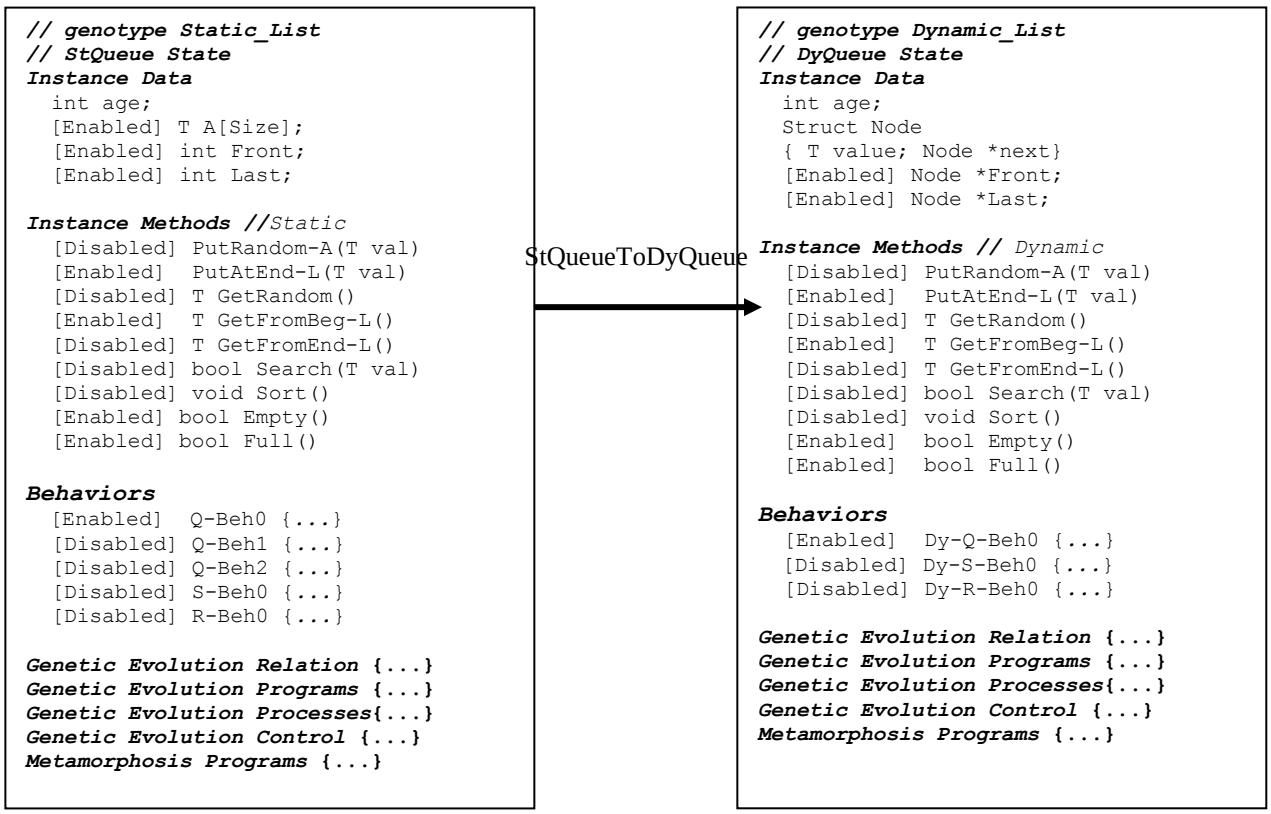
Funct_Id: The identifier of a user defined function ensuring the transition of specific persistent information from O1 to O2.

The metamorphose *StQueueToDyQueue* may be defined as it follows:

```
Metamorphosis_Program  StQueueToDyQueue
{
   Metamorphose to genotype Dynamic_List;
   At the Evolution State StQueue to the Evolution State
   DyQueue;
   Information transition ensured by the procedure
   StQueueToDyQueueTrans;
   procedure StQueueToDyQueueTrans
   {
      while (not StQueue.Empty( ))
      {DyQueue.PutAtEnd (StQueue.GetFromBeg( ) ;}
   }
}
```

Example 11: Metamorphosis program StQueueToDyQueue

Example 12 shows the translation of *StQueue* into a *DyQueue* done by the metamorphosis program *StQueueToDyQueue.*



Example 12: Translation of Queue from Static List into Dynamic List

The following application program creates an object *Q1* from the genome (class) List (Example 2), holding the features specified by the genotype *Static_List* (Example 3) and having the lifecycle defined by the GEProc *Static_Dynamic* (Figure 4-6)

{ …
List Static_List Q1= New (GEProc Static_Dynamic);
/ List is the genome class*
Static_List is a genotype of List, defining potential characteristics to be held by objects of this genotype.
Q1 is an instance (phenotype), inheriting its characteristics from List according to Static_List requirements.
GEProc Static_Dynamic is the GEProc defining the lifecycle of Q1(Figure 4-6).
*Q1 will have an initial state StQueue, defined by the GEProg StQueue (Example 8) and will behave according to the behavior Q-Beh0 (Figure 4-5) */

. . . // *Use of Q1 as static Queue*
}

Suppose an application program that uses the already created object Q1, at age = 10. The metamorphosis *StQueueToDyQueue* will be interpreted to metamorphose the Static_List Q1 to Dynamic_List Q1.

The interpretation of the genetic metamorphosis program *StQueueToDyQueue* will translate the precedent application program to an internal form as it follows:

```
{ …
  Static_List Q1; // Q1.age = 0
  Q1= New StQueue ( ); // Current GEProg is StQueue
   ….   // Use of Q1
   // At age= 10, Q1 should evolve into DyQueue
 // Generation of a new object Q1_New, of the Dynamic_List variation
 Dynamic_List Q1_New;
 Q1_New = New DyQueue( );.  // DyQueue is the target GEProg

 Q1_New.GEProc= Q1.GEProc; // Genetic information transfer
 // Specific information transfer
 Q1_New.age= Q1.age;
 While (not Q1.Empty( )) do  //
          { Q1_New.PutAtEnd-L  (Q1.GetFromBeg-L( ));}
    . . .        [1]      // Substitution of Q1 by Q1_New in [1].
}
```

Note that applications must always use objects according to the state of their actual Age.

4.4 Tolerating Evolution Accidents

Some environmental and internal factors cause different kind of accidents. A key element, in handling accidents, is detecting them at early stages. In order to achieve that, we are creating a checkpoint each time an object evolves. After detecting accidents, we should assess the damage caused by them, so that we can find a proper mechanism to repair it (if possible).

Accidents affecting the genetic evolution program

Some accidents may affect the genetic evolution program, causing some changes in it; i.e. an enabled element may become disabled and a disabled element may become enabled.

Example: Consider the following factor that affects StQueue genetic evolution program:

{Disable front;}

Disabling front disables GetFromBeg-L. This means that queue structure and operations are completely changed.

Accidents affecting the genetic evolution process

Some external factors could cause a failure in calling the GEProg at a certain point of time; this will cause the evolving object to remain at the same state without evolving to the next

state. Sometimes though the GEProg is successfully called, while being executed, some errors may occur and stop its execution.

Example: Consider the genetic evolution process in Figure (4-4).

Assume that at age 10, the GEProg StStack is damaged and could not be called. An object of this process will remain in StQueue state without evolving to StStack state.

Accidents affecting the Genetic evolution relations

Some accidents cause changes (update, deletion) in existing genetic relations. This kind of changes usually leads to an incoherency within predefined states or to new undefined states.

Example: Consider an environmental factor having the following effect on genetic evolution relations:

{Disable Front → Enable GetFromBeg-L;}

Disabling Front should disable GetFromBeg-L, but when this effect takes place, disabling Front will enable GetFromBeg- L.

Accidents affecting objects

This kind of accidents usually happens when the object is exposed to certain factors that cause the enabled structures/operations/behaviors to become disabled, or cause the disabled structures/functions/behaviors to become enabled. This leads to an incoherent state. Objects within this state cannot be used, until they are repaired.

In some cases, accidents do not only affect enabled/disabled elements within the object, but they affect the structures, operations and behaviors themselves; causing an error when these structures, operations and behaviors are being used without a proper repair.

Example: Consider an environmental factor having the following effect on an object in StStack state:

{Enable front;}

Enabling front enables GetFromBeg-L, Empty and Full. This is incoherency, since objects in a StStack state should not have front enabled as an attribute and GetFromBeg-L enabled as an operation.

How to detect an accident, assess the damage and repair it?

Each object evolution age represents a checkpoint for that object. At each checkpoint, a checking program will be running, whose task is detecting all kind of accidents that have been previously defined. If an accident is detected, a function will be running to determine

what parts of the object are damaged. At the end, another function will be running to repair the damage and eliminate the factors that caused the accident (not always applicable, since we cannot control and eliminate environmental factors).

In Figure (4-7), we take accidents that affect GEProg as an example, to show how accidents might be detected, then how the damage might be assessed and finally how the GEProg might be repaired.

```
PROGRAM Tolerate_Accidents
BEGIN

Struct Result {Bool exist; String name ;}
Declare Result A[ ];
 // Declare an Array of type Result
...
Call Detect ( ) // call Detect function
FOR i= 0 TO Length of A
    IF  A[i].exist = False THEN
    //If an accident is detected
       Damage=call Assess(A[i]);
       //Call the function Assess
       Call Fix(Damage)//Fix the damage
    END_IF
END_FOR
...// the rest of the program
END_PROGRAM
```

```
FUNCTION Detect( )
//This function detects accidents
BEGIN

X = Get current GEProg;
N = Get the name of current GEProg;
Y = Get the GEProg called N from the Genome;
...
FOR i = 0 TO Length of Y

  IF X's element equals its corresponding
  element in Y
  /* If the current GEProg matches its
  corresponding GEProg in the Genome */
  THEN
       A[i].exist = True;
       // No accident is detected
       A[i].name= "GEProg";
  ELSE
       A[i].exist =False;
       // An accident is detected
       A[i].name= "GEProg";
   END_IF
END_FOR
...// the rest of the Algorithm
END_FUNCTION
```

```
FUNCTION Assess(A[ ])
//This function locates the damage

BEGIN
   Read (A[i]);
   Return A[i].name //The name of the Damaged Part;
END_FUNCTION

FUNCTION Fix( )
//This function fixes the damage
BEGIN
IF Damage = GEProg THEN
       Get the Genome's copy of this GEProg;
       Replace Current GEProg with Genome's copy;
END_IF
... // the rest of the Algorithm
END_FUNCTION
```

Figure (4-7): Accidents tolerance algorithm, in pseudo code.

CHAPTER FIVE

EVALUATION: IMPLEMENTATION ISSUES, APPLICATION AREA AND COMPARISON

In this chapter, we evaluate the genetics-based methodology to object evolution, by introducing some issues related to implementation aspects, application area and a comparison.

5.1 Implementation Issues

The implementation environment of this methodology requires a strongly typed and an object-oriented programming language.

Strongly typed language. The programming language should ensure type-checking of objects each time an object is declared or created. The checking process should guarantee the correct association between the genome class and its genotypes, and between the genotypes and its objects (phenotypes), by keeping a record of the genome and its subclasses, and using it to check the genotype and the genome associated to a specific object.

Object-oriented programming language. We need to add an extension to any existing object-oriented programing language (OOPL), to adopt the notion of genome class, genotype, phenotype, GER, GEProc, GEProg and GEC.

We are building on an extension that was presented earlier in (S.Ghoul, 2011). These extensions might be processed by any OOPL preprocessor. Figure (5-1) illustrates our extension to an OOPL to implement the genome class, genotype, phenotype, GER, GEProc, GEProg and GEC. Objects can be created from a specific genotype as it follows:

```
Class<Id> genotype<Id> Object_name = New (GEPRoc GEProc<Id>);
```

When objects are created, they hold the same structure of their genotype. The interpretation process takes place at the creation of objects and each time objects evolve. This process fills *Enable set* and *Disable set* with structures and functions to be enabled/disabled. Once an element is added to *Enable set*, its access modifier, is set to one of the following: {public, protected, private}. We suggest a new access modifier, called *inactive.* Elements that are added to *Disable set* will have their access modifiers changed into *inactive.* Elements having this access modifier cannot be used until they are enabled.

```
Class Genome-Name {
    Config: (Name= Genotype_name)
    {
       Require
       {
          Structure(…);
          Function (…);
          Behavior (…);
       } //End Require
    }
   Control: (Name = GER)
   {
     Imply
     {
      Enable{Structures(…);Functions(…);}→Enable{Structures(…);Functions(…);}
      Disable{Structures(…);Functions(…);}→Disable{Structures(…);Functions(…);}
     }//End imply

     Exclude
     {
      Enable{Structures(…);Functions(…);}→Disable{Structures(…);Functions (…);}
      Disable{Structures(…);Functions(…);}→Enable{Structures(…);Functions(…);}
     } //End exclude

     Compound
     {
      Enable {Structures (…); Functions (…);} AND Disable {Structures (…);
      Functions (…);}→ Disable {Structures (…); Functions (…);} OR Enable
      {Structures (…); Functions (…);}
     }//End Compound
   } // End GER

   Control: (Name = GEProg)
   {
      Enable {
             Structures(…);
             Functions(…);
             }
      Disable {
             Structures(…);
             Functions(…);
             }
       }// End GEProg

    Control: (Name = GEProc)
    {
        Age = value : GEProg_i, Behavior_i;
    } //End GEProc

   Control: (Name = GEC)
   {
       … // GEC program
   } //End GEC

} // End Genome Configuration
```

Figure (5-1): An extension to an OOPL.

5.2 Application Area

Genetics-based evolution of objects can be used in many applications, where there is a real need for dynamic changes in objects at run time. Banking, GUI development, scientific and simulation programs are all examples of applications that usually require dynamic and continuous changes in their objects at runtime.

As an example, consider medical and scientific applications. In such applications, there is a need to simulate the lifecycle of some microscopic entities. These entities' lifecycle include many phases. At each phase, these entities change their structure and behavior completely. Figure (5-2) illustrates a lifecycle of a microscopic entity. It starts as an egg and evolves till it reaches an adult phase. When using our genetics-based approach, we will be able to simulate the growth process of these entities, using one object that represents the entity throughout its lifetime at different phases.

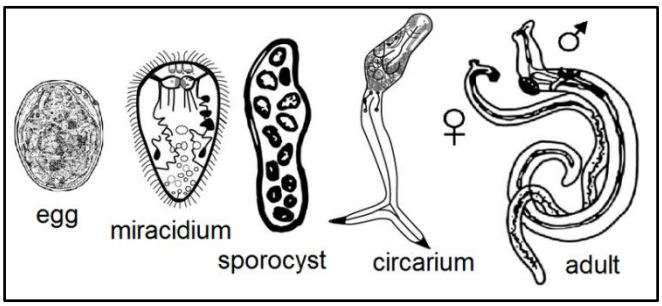


Figure (5-2): A life cycle of a microscopic entity.

Another example is GUI applications, which make use of different kind of shapes to create an animation. An object should evolve between different shapes (circle, rectangle, triangle, etc.). When following our genetics-based approach, there is no need to use multiple objects (one for each shape); since one object can evolve between different shapes.

5.3 Comparison

Since each object evolution approach has its own characteristics, in order to make a better comparison among them, we will define a set of evaluation criteria, based on our study to objects evolution:

- *Evolution inside a class:* The ability of objects to evolve between different states, within the same class. The evolution between two classes is supported by all the approaches mentioned earlier.
- *Evolution coherence control:* The availability of control over the evolution process. This control insures the coherence between object's structures and operations.
- *Gaining/losing structures:* the ability of objects to gain and lose structures.
- *Gaining/losing operations:* the ability of objects to gain and lose operations.
- *Behavioral evolution:* the ability of objects to change their behavior at runtime. i.e. change the sequence of functions execution.

- *Automatic evolution:* The ability of objects to evolve automatically at runtime, i.e. without the need to call a specific function or a procedure that is written within the application.
- *Accidents handling:* The ability to handle different kind of evolution accidents.
- *Close to real world concepts:* Distance between proposed concepts and real world concepts. i.e. how the evolution is done, do objects evolve the same way real entities do, etc. The closer to real world concepts, the better.
- *Classes and objects hierarchy:* Large or small hierarchies of classes and objects. The smaller the hierarchy, the better.

Table (5-1) shows a comparison (based on the above criteria) between our proposed approach and the previously studied ones. This comparison states clearly the value of our contribution.

Table (5-1): A comparison between object evolution approaches.

Criteria / Approach	Evolution inside a class	Evolution coherence control	Gaining/ losing structures	Gaining/ losing operations	Behavioral evolution	Automatic evolution	Accidents handling	Close to world concepts	Classes and objects hierarchy
Object Re-classification	**No**	**No**	**Gaining and losing**	**Gaining and losing**	**No**	**No**	**No**	**Far**	**Large**
Object evolution	**No**	**No**	**Only gaining**	**Only gaining**	**No**	**No**	**No**	**Far**	**Large**
Class evolution	**Yes**	**No**	**Gaining and losing**	**Gaining and losing**	**No**	**No**	**No**	**Far**	**Large**
Schema evolution	**Yes**	**No**	**Gaining and losing**	**Gaining and losing**	**No**	**No**	**No**	**Far**	**Large**
Dynamic roles	**Yes**	**No**	**Gaining and losing**	**Gaining and losing**	**No**	**No**	**No**	**Close**	**Large**
ORM	**No**	**No**	**Gaining and losing**	**No operations**	**No**	**No**	**No**	**Far**	**Large**
Bio-inspired objects evolution methodology	**Yes**	**Yes**	**Gaining and losing**	**Gaining and losing**	**Yes**	**Yes**	**Yes**	**Close**	**Small**

After comparing our approach with the existing approaches that are used in object evolution, we find that our approach introduces the following concepts:

- Evolution inside class (without modifying the class).
- Evolution coherence control.
- Behavioral evolution.
- Automatic evolution.
- Accidents handling.
- Small classes hierarchy.

CHAPTER SIX

CONCLUSION AND FUTURE WORKS

6.1 Conclusion

In this study, we have proposed a genetics-based approach to objects evolution, where objects can evolve at runtime automatically without changing their identity. Our proposed evolution approach is based on the concept of phenotype evolution. Phenotypes, which are similar to objects, are instances of genotypes, which hold the set of potential characteristics. Objects can lose/gain structures, operation and behaviors at runtime. To achieve this, we have added some genetics information to the genome class: GEProg, GEPrc and GEC.

Environment affects the evolution process and objects in different ways. In this work, we are presenting a mechanism to handle some kind of accidents caused by environmental factors. In order to implement our approach, we have presented a programming language extension to be added on any OOPL.

6.2 Future Works

- We have studied genetic-based objects evolution. This evolution is planned and pre-determined. A future work is to study evolution caused by environment, which includes learning (Epigenesis). Environmental factors may have huge influence on evolving objects, and they could cause huge differences between objects that belong to the same genotype.
- Reducing the time needed to create instances from genotypes, and the time needed for objects to evolve, using learning techniques.
- Studying the evolution of Genome, Genotype, GEProg, GER, GEProc and GEC.
- Formalizing our methodology using a formal specification language.

REFERENCES

Andrzej Jodloowski, Piotr Habela, Jacek Ploodzien. Dynamic Object Roles – Adjusting the Notion for Flexible Modeling, IDEAS '04 Proceedings of the International Database Engineering and Applications Symposium, IEEE, 2004, pp.449-456.

Artur Caetano, Marielba Zacarias, António Rito Silva, José Tribolet, (2005). A Role-Based Framework for Business Process Modeling, 38th Hawaii International Conference on System Sciences (HICSS-38), IEEE, 2005, Track 1 - Volume 01, Page 13.3.

Bertrand Meyer. Object-Oriented Software Construction, Second Edition, Prentice Hall, 2000.

Bruce Alberts, Alexander Johnson, Julian Lewis, Martin Raff, Keith Roberts, Peter Walter. Molecular Biology of the Cell , Fifth Edition , Garland Science, Taylor & Francis Group, 2008.

D.Ancona, C.Anderson, F.Damiani, S.Drossopoulou, P.Giannini, E.Zucca. A Provenly Correct Translation of Fickle into Java NEW Fickle, ACM Transactions on Programming Languages and Systems, April 2007, Volume 29 Issue 2.

Dareen Hammodeh. A Genetics-Based Approach to Inheritance Modeling. Master Thesis, Philadelphia University, 2012.

Djamel Meslati, Saïd Ghoul. Towards Autonomously Developed Software: A genetic Approach in Critical and Embedded Systems, Journal of Computer Sciences, 2005 Science Publications, volume 1, Issue 4 , pp. 530-537.

Dominik Stein, Stefan Hanenberg, Rainer Unland. Roles from an Aspect-Oriented Perspective, Roles, VAR'05 in conjunction with ECOOP, Glasgow, UK, 24 July, 2005.

Einar Broch Johnsen, Marcel Kyas, Ingrid Chieh Yu. Dynamic Classes: Modular Asynchronous Evolution of Distributed Concurrent Objects, FM '09 Proceedings of the 2nd World Congress on Formal Methods Springer-Verlag Berlin, Heidelberg, 2009, pp.596 – 611.

Ian Sommerville. Software Engineering, ninth edition, Addison Wesley, 2011.

Marco Piccioni, Manuel Oriol, and Bertrand Meyer. Class Schema Evolution for Persistent Object-Oriented Software: Model, Empirical Study, and Automated Support, IEEE Transactions on software engineering, Mar, 2011, volume: PP , Issue: 99. Page(s): 1.

Said Ghoul. Bio-inspired Systems – An Integrated Model. MISC2010, International Symposium on Modeling and Implementation of Complex systems, Constantine, Algeria,, 2010.

Said Ghoul. Supporting Aspect-Oriented Paradigm By Bio-inspired Concepts, ISIICT, 2011, IEEE, pp. 63 – 73, 2011

Tal Cohen, Joseph (Yossi) Gil*y*. Three Approaches to Object Evolution, PPPJ '09 Proceedings of the 7th International Conference on Principles and Practice of Programming in Java, ACM, pp. 57-66, 2009.

Terry Halpin. Object-Role Modeling -Principles and Benefits. International Journal of Information System Modeling and Design, 1(1), 2010, pp. 33-55.

منهجية مستوحاة من البيولوجيا لتطور الكائنات

بواسطة
إيناس توفيق سعد النفار

بإشراف
أ.د. سعيد الغول

قدمت هذه الرسالة استكمالاً لمتطلبات الحصول على درجة
الماجستير في علم الحاسوب

عمادة البحث العلمي والدراسات العليا
جامعة فيلادلفيا

أيار 2012

مـلـخـص

من إحدى القضايا المطروحة في البرمجة الشيئية (OOP) ، مسألة عدم وجود آليات لإعادة تصنيف الكائنات (object reclassification). إعادة تصنيف الكائن (object) تمكنه من تغيير فئته (class) أثناء تنفيذ البرنامج. عادة ما نحتاج الى إعادة تصنيف الكائنات في التطبيقات التي تتغير كائناتها بشكل دينامكي وقت تنفيذ البرنامج.

اعتمدت المناهج السابقة التي اتبعت في التعامل مع إعادة تصنيف الكائنات طرقا مختلفة. لكن لم يقترب اي منها من مفهوم تطور الكائنات البيولوجية ، وقد أدى ذلك إلى استخدام عدد كبير من الفئات ، الامر الذي جعل من تصميم وتنفيذ النظام أكثر تعقيدا.

ينبغي للتطور المنشود للكائن ان يسمح له بأن يتطور تلقائيا أثناء وقت تنفيذ البرنامج، من دون أي تغيير على فئته. في هذا العمل، نقدم منهجية جديدة لتطور الكائنات ، مستوحاة من بعض مفاهيم علم الوراثة. حيث يمكن للكائن الذي ينتمي إلى فئة محددة أن يغيير هيكله ووظائفه وسلوكه في وقت التشغيل تلقائيا، مع الحفاظ على انتمائه إلى نفس الفئة. تم إدخال تأثير طفيف للبيئية على عملية التطور المقترحة.