

# Design Patterns for Programmable Parameter Control for Evolutionary Algorithms

Mohammed Zubair

December 2009

Department of Computer Science  
California State University, Fresno

# Design Patterns for Programmable Parameter Control for Evolutionary Algorithms

Mohammed Zubair

December 2009

Department of Computer Science  
California State University, Fresno  
Fresno, CA 93740

Project Advisor  
Dr. Shih Hsi Liu.

Submitted in partial fulfillment of the requirements  
for the degree of Master of Science

## Abstract

Programmable Parameter Control for Evolutionary Algorithms (PPCEA) is a Domain Specific Language (DSL) specific for Evolutionary Algorithms (EA) domain. The underlying design and implementation of PPCEA semantics for Genetic Algorithms (GAs) and Evolution Strategies (ESs) is very complex. Software Engineering and Object-Oriented Principles (OOP) are not fully applied which results in complications for extension, changeability, and independent development. A modularized design and implementation of PPCEA that follows Software Engineering principles and the modularization advice from “On the Criteria” paper is required so that burden of future extension and evolution will be eased. Apart from design and implementation, an application that has an organized and user friendly Graphical User Interface (GUI) helps improve the overall usability and accessibility experience for users. The new PPCEA introduces a GUI to further improve the usability and accessibility to end users to reflect the cohesion of domain-specific languages and end user programming.

The new PPCEA is much more modularized, comprehensible and manageable: By applying design patterns and aforementioned principles, the new PPCEA has reduced a great number of dependencies (i.e., coupling) among the modules in ESs and GAs. Difficulties in modification and extension are greatly reduced compared to previous PPCEA. Additionally, independent development is uncomplicated, because of less dependency among different classes. Lastly, with the design of a GUI, end users can now interact with the newer PPCEA more efficiently.

## Preface

For the last few years a number of software tools for Evolutionary Algorithms (EAs) have been developed. PPCEA is powerful software which solves one of the major problems in the Evolutionary Algorithms domain, which is to control parameter settings of evolution algorithms which have significant impact on EAs performance. One of the problems with PPCEA is its design and implementation of Evolutionary Strategies and Genetic Algorithms are not fully object oriented which will result complications in maintenance and future extension.

The PPCEA needs to be redesigned to make it much superior. This report discusses how PPCEA design was greatly improved by following Software Engineering and object oriented principles. By following previously mentioned principles the design of PPCEA is enhanced, which will significantly reduce complications during maintenance and future development. Object oriented principles especially design patterns have been used in design of high quality software in diverse fields. Hence, various design patterns were used during the redesign and implementation of PPCEA. Additionally, to improve interactivities a graphical user interface is also designed.

With implementation of design patterns the newer PPCEA has many advantages compared to previous PPCEA. The new design was compared with the previous design using a software metrics tool available in Eclipse and the results show a significant reduction in complexity. Development and maintenance is uncomplicated because of fewer dependencies between modules. Also the implementation of Strategy and Chain of Responsibility patterns allows end users to select various algorithms implemented in ESs and GAs on the fly.

# CONTENTS

Abstract

Preface

1. Introduction.....	1
2. Evolution Algorithms.....	4
3. Domain Specific Languages.....	6
4. PPCEA.....	7
5. Software Engineering Principles and Design patterns.....	10
6. Software Engineering Principles and Design patterns in PPCEA.....	12
6.1 Visitor Pattern.....	13
6.2 Decorator Pattern.....	18
6.3 Strategy Pattern.....	22
6.4 Singleton Pattern.....	26
6.5 Composite Pattern.....	28
6.6 Chain of Responsibility Pattern.....	31
7. Software Metrics.....	35
8. Graphical User Interface for the PPCEA.....	41
9. Related Work.....	43

10. Conclusion.....45

Bibliography.....46

## LIST OF FIGURES

Figure	Page
1. Procedure Evolution Algorithm	5
2. PPCEA Framework	8
3. Denotational Semantics of PPCEA	9
4. UML Diagram of Visitor pattern i	14
5. UML Diagram of Visitor Pattern ii	15
6. Accept method implementation	15
7. Implementation of CrossOverVisitor class	16
8. Executing crossover operation	16
9. UML Diagram of Decorator pattern implementation	19
10. EntropyDecorator class implementation	20
11. RoscaEntropy class implementation	21
12. Example calculation of Entropies	21
13. UML Diagram of Strategy pattern implementation	23
14. ContextCrossover class implementation	24
15. Executing default crossover strategy	24
16. Changing a strategy	25
17. Structure of Singleton Pattern	26
18. Implementation of GetEvaluateInstance method	27
19. Structure of Composite pattern	29
20. Chain of responsibility flow	31

21. UML Diagram of Chain of Responsibility pattern implementation	32
22. Implementation of CORStrategy class	33
23. Initialized chain of crossover algorithms	33
24. Calling successor in the chain	34
25. Metrics of CallGAStatement in PPCEA	37
26. Metrics of CallGAStatement in PPCEA 1.0	38
27. Metrics of CalLESStatement in PPCEA	39
28. Metrics of CalLESStatement in PPCEA 1.0	39
29. GUI for PPCEA 1.0	41
30. GUI for PPCEA 1.0	42

## 1. Introduction

Programmable Parameter Control for Evolutionary Algorithms (PPCEA) [1] is a Domain Specific Language (DSL) [7] specific for Evolutionary Algorithms (EA) [2] domain. The underlying design and implementation of PPCEA semantics for Genetic Algorithms (GAs) [2] and Evolution Strategies (ESs) [2] is very complex and lacks modularity. Namely, a number of Software Engineering and Object-Oriented Principles (OOP) (e.g., Open-Closed Principle [8], Liskov Substitution Principle [9] and Dependency Inversion Principle [10] as well as design patterns [5]) are not fully applied which results in complications for extension, changeability, and independent development. Apart from design and implementation, an organized and user friendly Graphical User Interface (GUI) for PPCEA is required to improve the usability and accessibility experience for users.

A modularized design and implementation of PPCEA that follows aforementioned principles and the modularization advice from “On the Criteria” [16] paper is required so that burden of future extension and development will be eased. Software Engineering principles are essential concepts required to build high-quality software. OOP principles especially design patterns will play an important role in redesigning the structure of PPCEA. Design patterns are solutions to common occurring problems in software development that help to reduce software complexity and make software comprehensible. Design patterns inherently follow software engineering principles and result in reusable, flexible, extensible and manageable object oriented design.

By using Java Foundation Classes (JFC) [4], an easy to use and powerful GUI for PPCEA needs to be designed. JFC provides features to build graphical user interfaces to improve interactivity to applications.

The newer design of PPCEA from now referred as PPCEA 1.0 is much more modularized, comprehensible and manageable: By applying design patterns and aforementioned principles, PPCEA 1.0 has reduced complexity, great number of dependencies (i.e., coupling) among the modules in ESs and GAs. The results from Eclipse software metrics tool have shown significant reduction in complexity of PPCEA 1.0. Difficulties in modification and extension are greatly reduced compared to previous PPCEA. Additionally, independent development is uncomplicated, because of less dependency among different classes. PPCEA 1.0 implements a number of algorithms for different kinds of operations, with the implementation of Strategy and Chain of responsibility patterns, clients will be able to select those algorithms dynamically, which promote achievements of even better fitness results and faster convergence rates.

Lastly, with the design of a GUI, end users can now interact with PPCEA 1.0 more efficiently. The GUI reflects the cohesion of PPCEA 1.0 semantics implementation of GAs and ESs. PPCEA 1.0 implements a number of algorithms for performing various operations, one of the important features implemented in design of GUI is to provide end users options to select different algorithms easily.

This report is organized as follows: Chapters 2 and 3 introduce Evolutionary Algorithms and Domain-Specific Languages, respectively; Chapter 4 provides a brief explanation of PPCEA; Chapter 5 provides brief explanation of Software Engineering principles and design patterns; Chapter 6 illustrates how design patterns and Software

Engineering principles were used to realize PPCEA 1.0; Chapter 7 provides the results obtained from Eclipse software metrics tool; Chapter 8 gives information of the design of GUI for PPCEA 1.0; Chapter 9 summarizes related work; and Chapter 10 addresses conclusion.

## 2. Evolutionary Algorithms

During the last few years there has been a growing interest in problem solving systems based on the principles of evolution and hereditary: Such systems maintain a population of potential solutions, the systems have some selection processes based on fitness of individuals and genetic operators [2].

Evolutionary Strategies and Genetic Algorithms are parts of Evolutionary Algorithms implemented in the PPCEA. Evolution Strategies are algorithms which imitate the principles of natural evolution for parameter optimization problems [2]. Genetic Algorithms are stochastic algorithms whose search methods model some natural phenomena: genetic inheritance and Darwinian strife for survival [2].

The structure of an evolution program taken from [2] is shown in Figure 2-1. As shown in Figure 2-1, all Evolution Algorithms maintain a population of individuals, which are possible solutions to the given problem and implemented as a data structure. The population undergoes alteration by selection, crossover and mutation. After each iteration, a new population is formed by selection of individuals from the population. The final population obtained is near optimum solution to the problem.

## Procedure Evolution Algorithm:

```
Begin
  t <- 0
  Initialize P(t)
  Evaluate P(t)
  While (not termination Condition) do
    begin
      t <- t+1
      Select P(t) from P(t-1)
      Alter P(t)
      Evaluate P(t)
    End
  End
End
```

Figure 2-1 – Procedure Evolution Algorithm [2].

### 3. Domain Specific Languages

Domain-Specific Languages (DSLs) are languages tailored to a specific application domain, and they offer substantial gains in expressiveness and ease of use compared with general-purpose programming languages in their domain of application [7]. A DSL is different from a general-purpose language (e.g., C++ and Java), because it performs specific tasks in a particular problem domain. For example, it can be a language that describes a business process (e.g., BPEL [23]), a database (e.g., SQL [24]), and evolutionary algorithms (e.g., PPCEA), among others.

Development of a DSL can be valuable if the language allows a particular type of problems or solutions to them to be expressed more clearly compared to other language. If the domain knowledge is to be expressed clearly in a programming language, a DSL is necessary. Appropriate domain specific notations are usually beyond the limited user-definable operator notations offered by general purpose languages. A DSL offers appropriate domain-specific notations from the start [7]. DSL's grammar matches the problem domain, not the base language's standard grammar which reduces the programming expertise required. As DSL's provide a lot more clarity in the domain, domain users can easily understand, modify and develop DSL programs.

#### 4. Programmable Parameter Control for Evolutionary Algorithms (PPCEA)

PPCEA is a DSL which is specific to Evolutionary Algorithms domain and its core code is written in Java. PPCEA was developed to solve the problems to control parameter settings in a programmable fashion. It keeps the Evolutionary Algorithm simple and lifts the problems of control parameter settings into a higher abstraction layer by using meta-programming [1]. The framework of PPCEA interpreter is shown in Figure 4-1. In the figure, JLex is lexical analyzer generator which breaks an input stream of characters into tokens. Constructor of Useful Parsers (CUP), implemented in Java, is a package for generating parsers. JLex and CUP define the grammar of PPCEA which consists of specialized statements (e.g., CallES and CallGA) and common linguistic elements (e.g., if-then-else statement) of imperative programming languages [1].

CUP defines how PPCEA's interpreter executes/interprets the source code of PPCEA, initializing from the bottom of the parse tree of PPCEA's source code. CUP traces up to check the syntax and executes the semantics [1]. As CUP meets EA statements and common linguistic elements, PPCEA interpreter processes the corresponding operations. The fitness function file at the top of the framework defines the fitness function, and the other files are different approaches for performing evolutionary processes. The domain-specific parameters that are generated or updated by the PPCEA interpreter (e.g., Best, Worst, and Average) are shown at the bottom of the framework. The Denotational Semantics of PPCEA are shown in Figure 4-2.

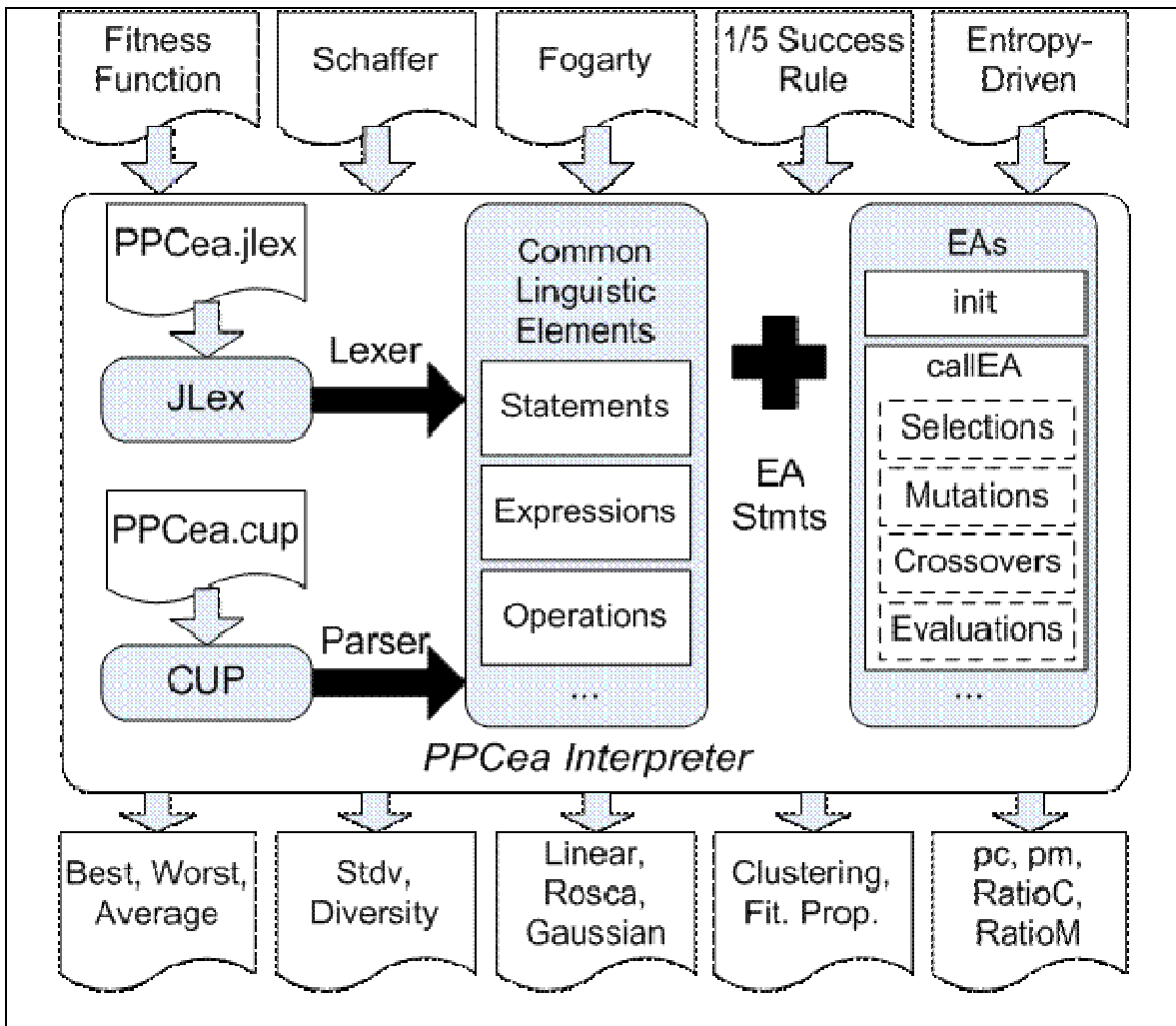


Figure 4-1 – PPCEA Framework [22].

## Denotational Semantics of PPCEa

### Abstract Syntax:

#### Syntax Domains: Abstract Syntax Rule

```
P: Program      P ::= genetic S end genetic
S: Statements   S ::= S1 ; S2 | read Ids | write Ids |
                if C then S fi |
                if C then S1 else S2 fi |
                while C do S end | ID assign E |
                init | callEA | readfile FileName |
                writeresult | validateMsg |
                invalidateMsg
C: Comparison   C ::= ( E R E )
E: Expression   E ::= E1 O E2 | ( E ) |
                N | D | ID
O ::= + | - | * | /
R ::= < | > | <= | >= | != | ==
N ::= Integer
D ::= Double
FileName ::= String
```

(a) Syntax

Figure 4-2 Denotational Semantics of PPCEA [21].

## 5. Software Engineering Principles and Design Patterns

There are a number of issues that software developers face today during the design and development of software because of its complexity. Complexity is inherent software [15], because there are a number of components and interactions between them as well as components' states before and after interactions. A good design of software will help in managing, developing and maintaining of software effectively.

Software Engineering principles describe properties of software, and describe methods and techniques for best development of software. Some important principles of software engineering are separation of concerns [17], abstraction, open-closed principle [8], Liskov substitution principle [9], dependency inversion principle [10], among others. This report discusses many techniques/principles that were followed to make PPCEA much better.

Design patterns are a part of the cutting edge of the object oriented technology [5]. Design Patterns are solutions to common occurring problems in software design. Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that programmers can use this solution a million times over, without ever doing it the same way twice [6]. Patterns do not give the code. Instead, they guide programmers how to design systems with good object oriented design qualities.

Each design pattern recognizes a problem, proposes solution and shows the consequences. Design patterns can be categorized into the following three groups in terms of the underlying problem they solve.

#### a. Creational Patterns

As the name suggests, these patterns deal with the creation of objects. For instance, the Singleton pattern restricts the instantiation of a class to a single object. Other examples of creational patterns are Abstract Factory, Builder, and Prototype.

#### b. Structural Patterns

These patterns describe ways to assemble objects. Structural Patterns include Adapter Bridge, Composite, Decorator, Facade, Flyweight, and Proxy.

#### c. Behavioral Patterns

These patterns deal with the flow control and algorithms. Chain of Responsibility, Command, Iterator, Mediator, Memento, Observer, State, Strategy and Visitor are within this category.

## 6. Software Engineering Principles and Design Patterns in PPCEA

A number of researchers used design patterns for designing software in diverse fields (e.g., control systems [11], state search research application [12], CORBA systems [13], Java framework for evolutionary algorithms (JCLEC) [14]) to improve and effectively design software. To improve the structure of PPCEA, the following design patterns were applied: (1) Visitor; (2) Strategy; (3) Composite; (4) Chain of Responsibility; (5) Decorator; and (6) Composite.

Software Engineering principles such as open closed principle, Liskov substitution principle and dependency inversion principle are inherently followed in any design pattern.

The Open Closed Principle states software entities should be open for extension and closed for modification [8].

The Dependency Inversion Principle is the idea to depend on abstract classes or interfaces instead of concrete classes [9].

Liskov substitution principle states inheritance should ensure that any property proved about super type objects also holds for subtype objects [10]. Functions that use pointers or references to base classes must be able to use objects of derived classes without knowing it [10].

The following subsections discuss in details about the design patterns that are applied to PPCEA.

## 6.1 Visitor Pattern

Visitor pattern lets programmers define a new operation without changing the classes of the elements on which it operates. Related behavior is not spread in different classes that define the object structure. Instead it is localized in a single visitor class.

CallESStatement and CallGASStatement classes in PPCEA have many methods in common (e.g., Crossover, Mutate, Getstat, SelectParents and SelectSurvivors). By implementation of Visitor pattern, all these individual methods in different classes can be localized in a single visitor class for each method.

The UML Diagram is shown in Figure 6-1, StatementVisitor is an abstract class and all the visitor classes (e.g., CrossoverVisitor, MutateVisitor) inherit this abstract class. The CrossOverVisitor, for example, is a class which has crossover operation of both CallES and CallGA statements. To use this visitor class the CallESStatement and CallGASStatement classes have to be added with an *accept* method which accepts a visitor and then the visitor calls back the appropriate method to be executed in the visitor class. The UML Diagram of CallESStatement and CallGASStatement is shown in Figure 6-2, both classes inherit abstract Statement class. And Figure 6-3 shows the implementation of *accept* method in CallGASStatement.

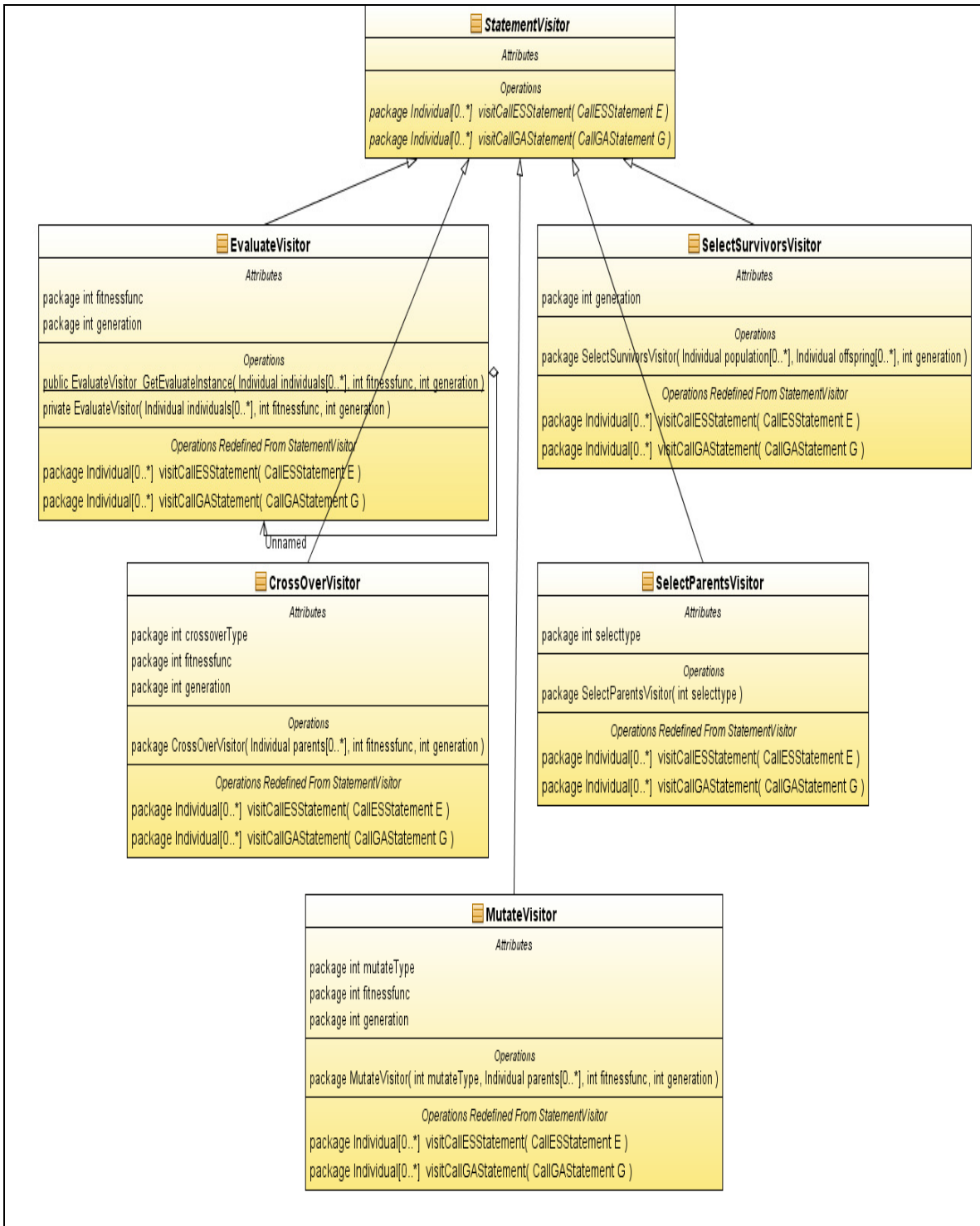


Figure 6-1 – UML Diagram of Visitor Pattern (i).

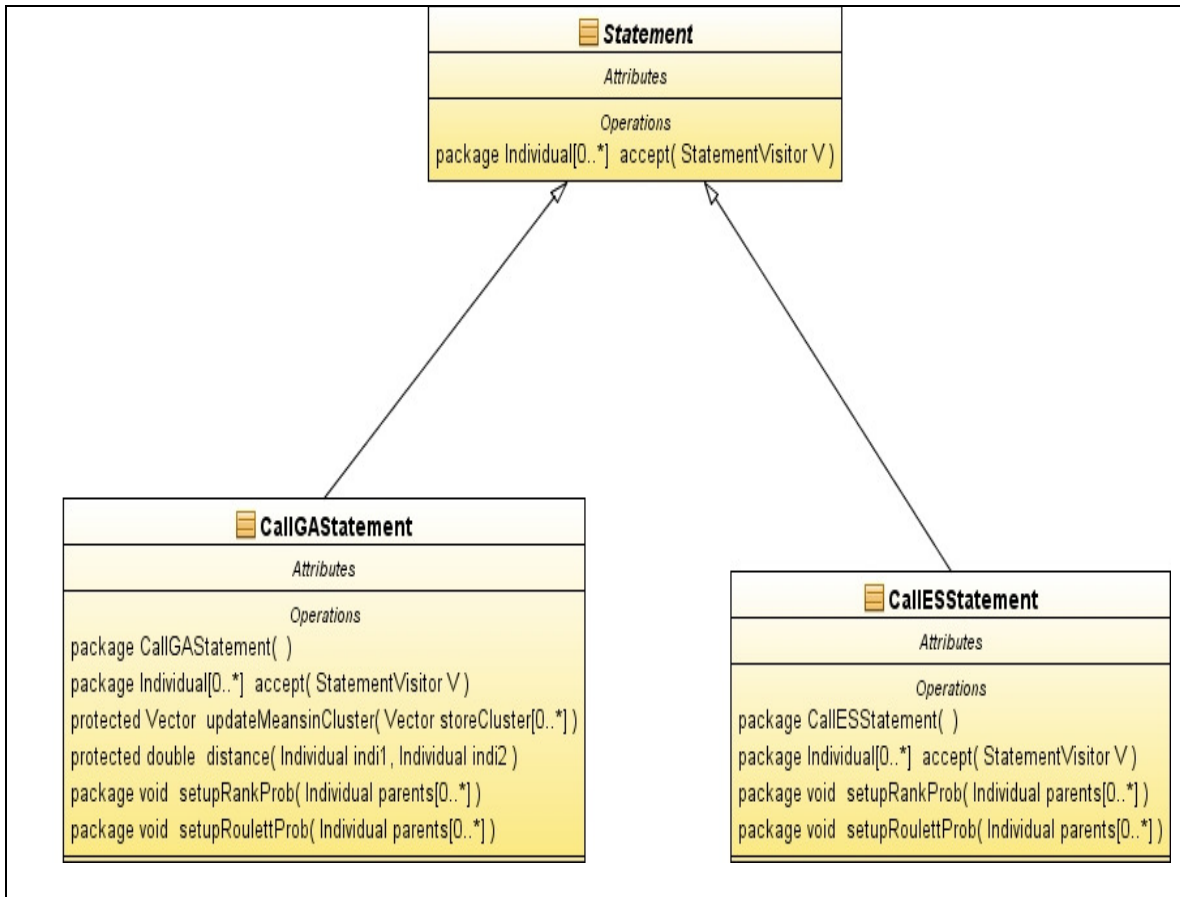


Figure 6-2 – UML Diagram of Visitor pattern (ii).

```

CallGASStatement
{
    Individual [] accept (StatementVisitor V)
    {
        V.visitCallGASStatement (this);/*callback the visitor*/
        /* "this" is a keyword in Java which represents the current object.*/
    }
}
  
```

Figure 6-3 – accept method implementation.

As stated earlier the visitor classes implement operation of both CalLES and CallGA. Figure 6-4 shows the implementation of CrossoverVisitor class, for example.

```
CrossOverVisitor extends StatementVisitor {  
  
    Individual [] visitCalLESStatement (CalLESStatement E)  
  
    {  
  
        Crossover Implementation for CalLES  
  
    }  
  
    Individual [] visitCallGASStatement (CallGASStatement G)  
  
    {  
  
        Crossover Implementation for CallGA  
  
    }  
  
}
```

Figure 6-4 Implementation of CrossOverVisitor class.

To use CrossOvervisitor class, the client creates a visitor object and passes it as an argument to *accept* method. The visitor object calls back the appropriate method. For example, to execute the Crossover operation, the code in Figure 6-5 needs to be written.

```
StatementVisitor V = new CrossOverVisitor (arguments);  
  
accept(V);
```

Figure 6-5 – Executing crossover operation.

The *accept(V)* method calls back the Crossover Operation based on the type of object in the CrossoverVisitor class.

PPCEA had all the unrelated operations in a single class. Now with the implementation of visitor pattern, all related operations are in different subclasses of visitor, which makes the PPCEA 1.0 more comprehensible and manageable. Addition of new operations is easy. A new operation can be added by simply adding a visitor class. As all the methods have their own visitor classes, they can be changed or developed individually without any changes to CallES and CallGA classes.

## 6.2 Decorator pattern

Decorator Pattern is sometimes also known as the Wrapper. The Decorator pattern attaches additional responsibilities to an object dynamically. Decorators provide a flexible alternative to sub-classing [3].

The Decorator pattern provides a more flexible way to add responsibilities to objects than can be had with inheritance. Inheritance requires creating a new class for each additional responsibility, and this may result in class explosion and increases the complexity of the software.

Different entropies are calculated in Getstat class, entropy is a concept in thermodynamics, information theory, statistical mechanics, and other fields to express disorder [18]. Entropy in information theory has to do with how much randomness there is in a signal or random event [18]. Getstat class calculates the following entropies: Linear [21], Clustering [21], Gaussian [21], Rosca [21], and Fitness Proportional [21]. Sometimes the calculation of all these entropies might not be necessary. The decorator pattern can be used here, as the intent of the Decorator pattern is to add responsibility to an object dynamically. With Decorator pattern various entropy calculations can be added and removed simply by attaching and detaching them.

The UML diagram of the implementation is shown in Figure 6-6. As shown in the UML Diagram the implementation involves encapsulating the original object (LinearEntropy object) inside an abstract wrapper interface (Entropy). Both the decorator classes (e.g., Cletropy and RoscaEntropy) and the core class (LinearEntropy) inherit from this abstract interface. The interface uses recursive composition to allow any number of decorators (entropies) to be added to core object.

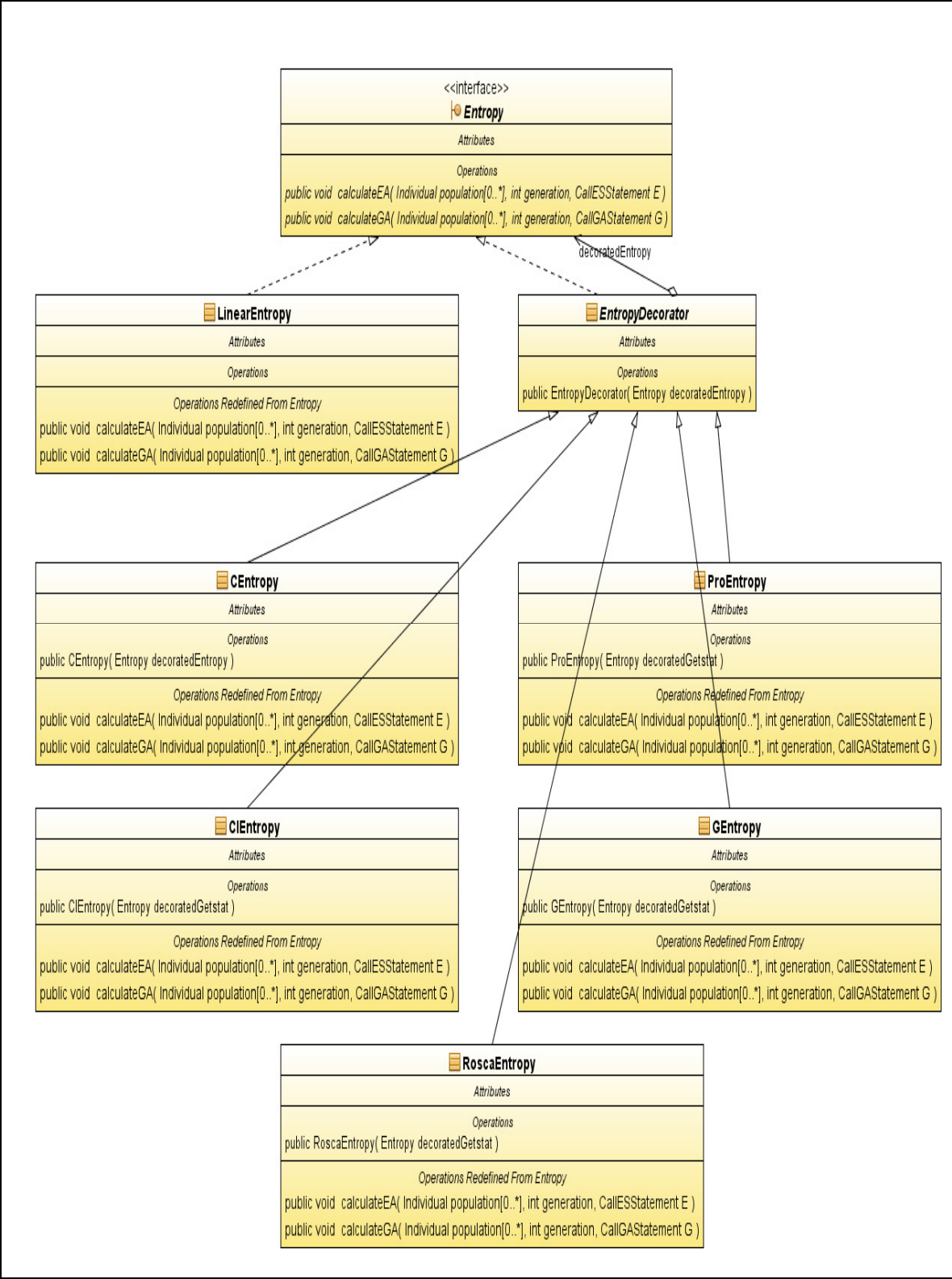


Figure 6-6 UML Diagram of Decorator pattern implementation.

Classes which inherit Decorator class always delegate to the Decorator base class and that class always delegates to the contained “wrappee” object. The “decorator” class EntropyDecorator implements Entropy interface and the implementation of decorator (EntropyDecorator) class is shown in Figure 6-7.

```
public abstract class EntropyDecorator implements Entropy
{
    protected Entropy decoratedEntropy; // the Entropy object being decorated

    public EntropyDecorator (GetstatBehavior decoratedEntropy)
    {
        this.decoratedEntropy = decoratedEntropy;
    }
}
```

Figure 6-7 EntropyDecorator class implementation.

As shown in the above code snippet, the EntropyDecorator class has a reference to the Entropy object being decorated. The classes which define additional responsibilities (e.g., Clentorpy and Rosca) inherit from the EntropyDecorator class. And implementation of RoscaEntropy for example is shown in Figure 6-8.

```
public class RoscaEntropy extends EntropyDecorator
{
    public void calculateGA(arguments)
    {
```

```

    decoratedEntropy.calculateGA (arguments);//delegation to base class

    AddedBehaviour;// add operation of RoscaEntropy calculation
}
}

```

Figure 6-8 RoscaEntropy class implementation.

The LinearEntropy class implements the Entropy interface and defines the core operation which is always executed, which is to calculate linear entropy in our case. The other subclasses of the EntropyDecorator which implement operations of other entropy calculation can be added or removed. For example, Figure 6-9 shows how only three entropies are calculated.

```

EntropyDecorator decorator =
        new ProEntropy(new RoscaEntropy((new LinearEntropy()));
decorator.calculateGA(arguments);

```

Figure 6-9 example calculation of entropies.

PPCEA calculated all the entropies every time. The Decorator pattern implementation in PPCEA 1.0 gives end users the ability to specify whatever combination of entropy calculation is desired and reduces the execution time of PPCEA 1.0, compared to PPCEA.

## 6.3 Strategy Pattern

The intent for this pattern is to define a family of algorithms, encapsulate each one, and make them interchangeable [3].

Different kinds of crossover/mutation/selection algorithms are implemented in PPCEA. Implementation of PPCEA used complex switch statement to select and execute those algorithms. A better approach is to use Strategy pattern which allows client to select algorithms dynamically.

The Strategy Pattern has a context class, a strategy class and concrete strategy. The UML diagram of Strategy pattern implementation of different crossover algorithms is shown in Figure 6-10. As shown in the following figure, the Strategy is an interface, which defines the method *execute()*, but does not implement it. All different strategies or algorithms implement the Strategy interface.

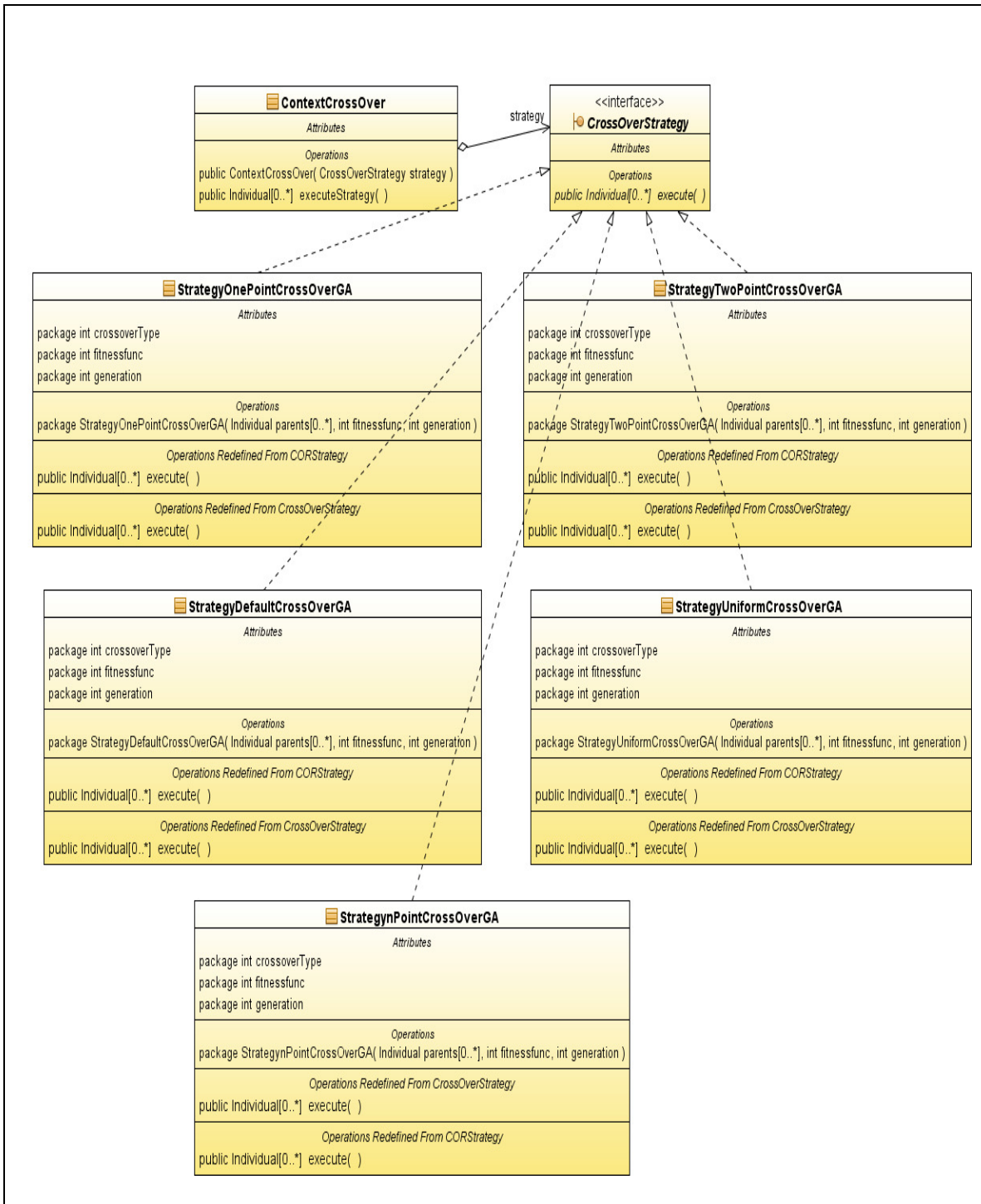


Figure 6-10 UML Diagram of Strategy pattern implementation.

The client interacts with the ContextCrossOver class, the implementation of the ContextCrossOver class is shown in Figure 6-11.

```
class ContextCrossOver {  
  
    private CrossOverStrategy strategy;  
  
    public ContextCrossOver (CrossOverStrategy strategy)  
    {  
        this. strategy = strategy  
    }  
  
    public Individual[] executeStrategy()  
    {  
        return strategy.execute();  
    }  
  
}
```

Figure 6-11 ContextCrossover class implementation.

Figure 6-12 shows the usage of ContextCrossOver class, for example, to execute DefaultCrossover algorithm for CallGA class.

```
ContextCrossOver context;  
  
context = new ContextCrossOver(new StrategyDefaultCrossOverGA(arguments));  
  
context.executeStrategy();
```

Figure 6-12 Executing default crossover strategy.

If end users need to change the strategy, then a new strategy needs to be assigned to the context as shown in Figure 6-13.

```
Context = new ContextCrossOver(new StrategyOnePointCrossOverGA(arguments));  
context.executeStrategy();
```

Figure 6-13 Changing a strategy.

With the implementation of Strategy pattern in PPCEA 1.0, end users can select and change algorithms dynamically. The semantics to select different strategies can be implemented at DSL level and users can write programs to select strategy on the fly. The use of complex if-then-else and switch statements is not required, because Strategy pattern eliminates conditional statements by encapsulating each algorithm in a strategy. Addition of new algorithms will be uncomplicated as new operation will be a subclass of Strategy class. As different algorithms have their own classes, development and modifications to them will be easier.

## 6.4 Singleton Pattern

The intent of Singleton pattern is to ensure a class has one or a countable number of instances and provide a global point of access to it. This pattern provides controlled access to instance and permits refinement of operations and representation [19]. Singleton objects do not pollute the global namespace (or, in languages with namespaces, their containing namespace) with unnecessary variables [5].

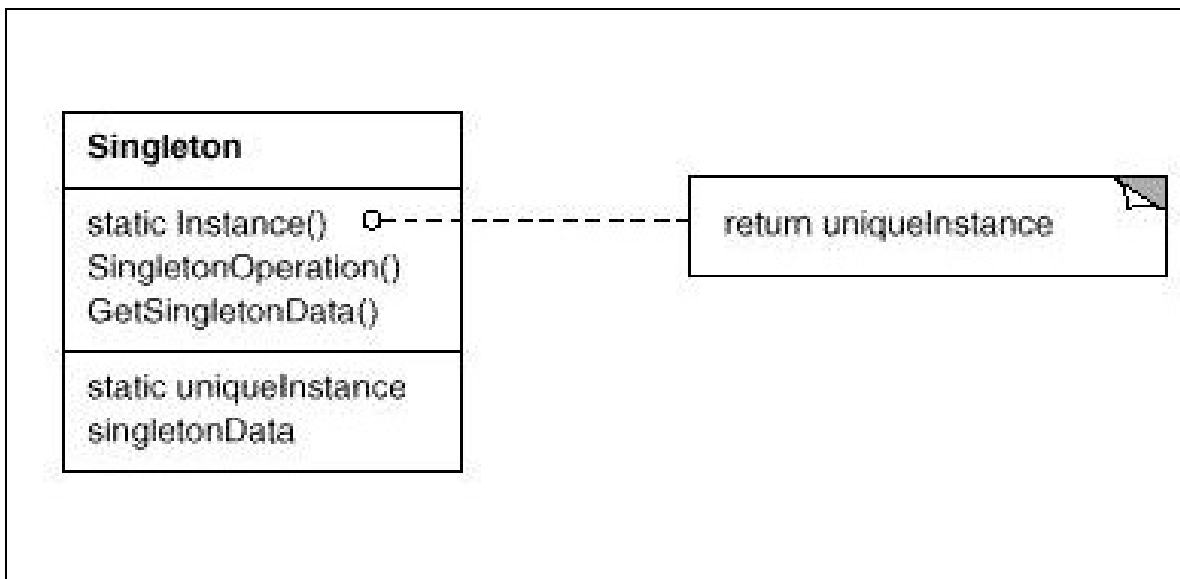


Figure 6-14 Structure of Singleton Pattern [20].

As mentioned earlier the `Getstat` class in PPCEA calculates different entropies. `Getstat` class needs to be instantiated only for the number of individuals. Singleton pattern can be used to restrict the number of instantiations.

The constructor of the class for which singleton has to be implemented is made private. The only way to instantiate the class is to use the static *Instance* method provided

as shown in Figure 6-14. Once it is instantiated, the same unique instance is always returned.

```
public static GetEvaluateInstance(arguments)
{
    if(Parameters.count<number of individuals)
    {
        uniqueInstance = new EvaluateVisitor(arguments);
        Parameters.Count++
    }
    return (uniqueInstance);
}
```

Figure 6-15 Implementation of GetEvaluateInstance method.

Figure 6-15 shows the implementation of GetEvaluateInstance method of EvaluateVisitor class for example. This *GetEvaluateinstance* method instantiates and allows only a countable number of instances to be created. As Getstat and Evaluate classes need to be instantiated only for the number of individuals present in the Evolution Algorithm process, PPCEA 1.0 implements Singleton pattern to restrict the number of instantiations.

## 6.5 Composite Pattern

The intent of composite pattern is to compose objects into tree structures to represent whole-part hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.

The PPCEA semantics consists of specialized statements such as CalLES and CallGA and common linguistic statements such as if-then-else and while statements. The clients should be able to treat composite objects (e.g., while statement) which can have statements inside and individual objects (e.g., CalLES) uniformly.

The structure of Composite pattern is shown in Figure 6-16. Abstract Component class (Statement) that specifies the behavior that needs to be exercised uniformly across all leaf and composite objects. The leaf classes (e.g., CalLES, CallGA) and Composite classes (e.g., WhileStatement) inherit from the component (Statement) class. Each Composite object “couples” itself only to the abstract type component (Statement) as it manages its “children”.

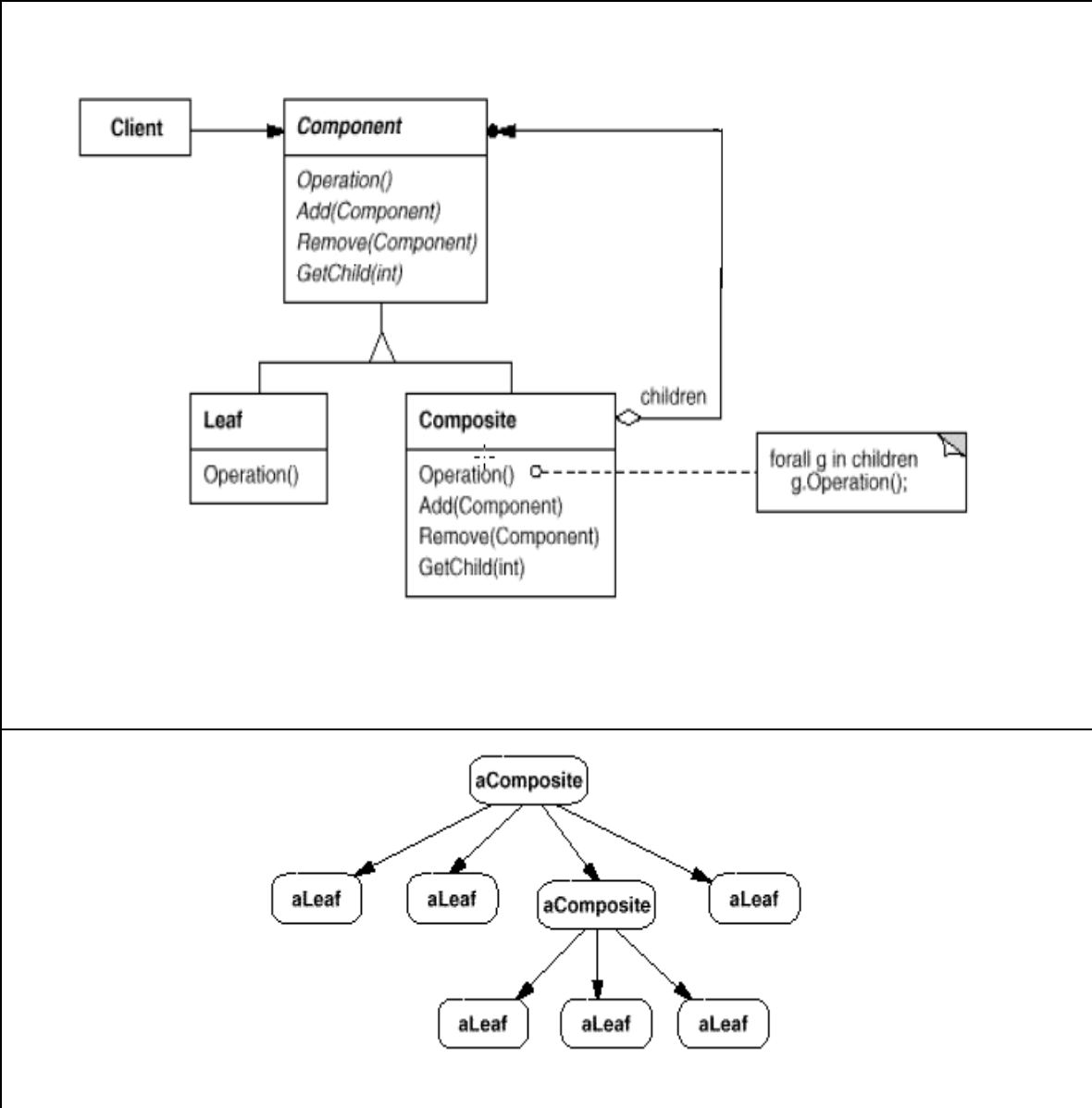


Figure 6-16 Structure of Composite pattern [20].

With the implementation of Composite pattern in PPCEA 1.0, any new statements in future can be added without complications, as a new statement will be a subclass of abstract Statement class. This pattern follows open-closed principle, and in the future the development of statement class will be easier as the clients remain unchanged and do not have to know if they are dealing with composite or a leaf component.

## 6.6 Chain of Responsibility

There could be a requirement to select algorithms based on some conditions in many cases. The intent of Chain of Responsibility pattern is to put algorithms in a chain and pass the request along the chain of handlers until an object handles it, as shown in Figure 6-17. The object that initiates the request does not know the object that will eventually provide the help.

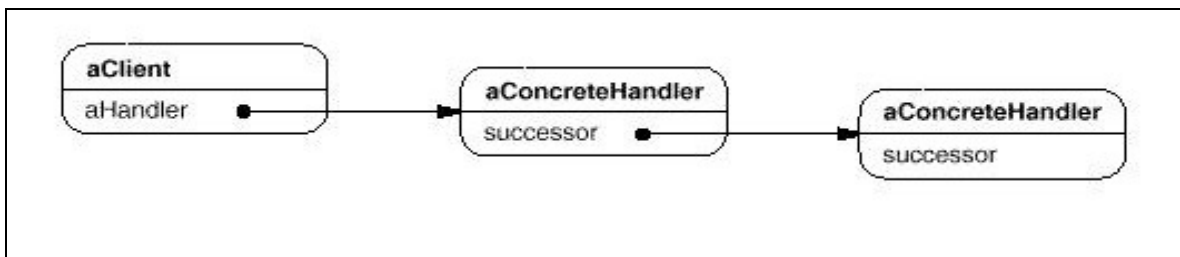


Figure 6-17 Chain of responsibility flow [20].

In PPCEA, the CrossOverVisitor class implements a number of algorithms for different crossover operations (e.g., one point crossover, two point crossover, and uniform crossover etc). There should be an option to select an algorithm dynamically during execution based on some conditions. Chain of responsibility pattern can be used to decide the algorithm to be executed dynamically. UML Diagram for chain of responsibility is shown in Figure 6-18.

All the algorithms inherit the abstract CORStrategy class and define the *execute* method (different algorithms). Whenever a request is received, it is handled if the condition satisfies. Otherwise, it must be forwarded to another object to handle the request.

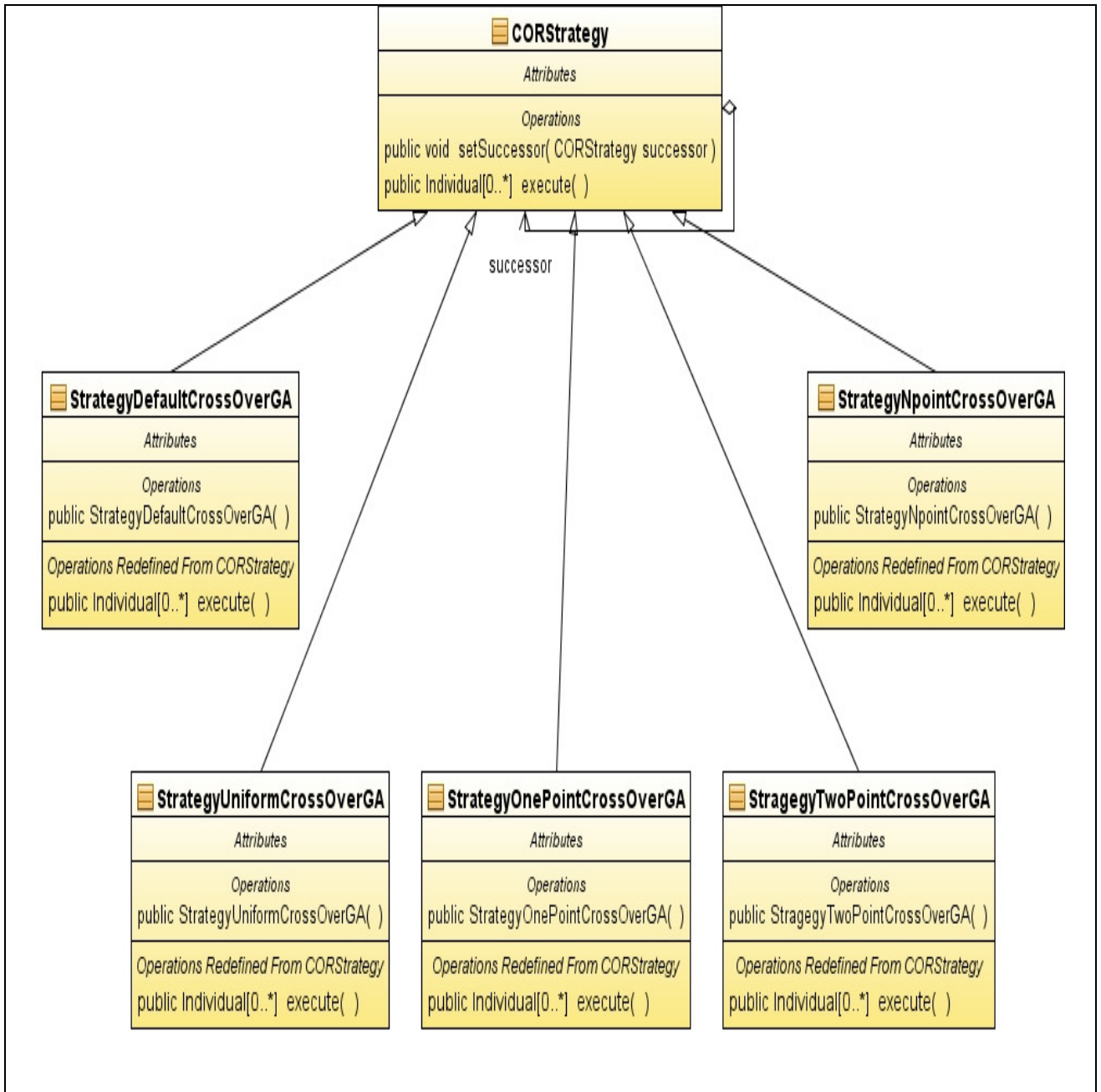


Figure 6-18 UML Diagram of Chain of Responsibility pattern implementation.

The request has to be passed along some chain of handlers, and the setSuccessor method is implemented to create the successor in the chain. Figure 6-19 shows the implementation of CORStrategy class.

```

public abstract class CORStrategy {

    protected CORStrategy successor;

    public void setSuccessor(CORStrategy successor)

    {

        this.successor = successor;

    }

    abstract Individual[] execute();

    }

```

Figure 6-19 Implementation of CORStrategy class.

The request is passed along the initialized chain, for example, for crossoverGA shown in Figure 6-20, and this chain of handlers can be modified dynamically.

```

StrategyDefaultCrossOverGA defaultstrategy = new StrategyDefaultCrossOverGA();

StrategyUniformCrossOverGA uniformstrategy = new StrategyUniformCrossOverGA();

StrategyOnePointCrossOverGA onepointstrategy = new StrategyOnePointCrossOverGA();

StrategyTwoPointCrossOverGA twopointstrategy = new StrategyTwoPointCrossOverGA();

StrategynPointCrossOverGA npointstrategy = new StrategynPointCrossOverGA();

defaultstrategy.setSuccessor(uniformstrategy);

uniformstrategy.setSuccessor(onepointstrategy);

onepointstrategy.setSuccessor(twopointstrategy);

twopointstrategy.setSuccessor(npointstrategy);

defaultstrategy.execute();

```

Figure 6-20 Initialized chain of crossover algorithms.

Every time the request passes along the chain, the condition is checked. If it satisfies then a specific strategy is executed, or it is passed to its successor, as shown in Figure 6-21.

```
if( successor != null)
return successor.execute();
```

Figure 6-21 Calling *successor* in the chain.

PPCEA 1.0 implements this pattern so that algorithms can be selected dynamically based on some conditions. The semantics to select different chain of strategies can be implemented at DSL level and users can write programs to select any pre-defined chain on the fly. The old PPCEA used complex if-then-else and switch statements to execute the algorithms, and the implementation of all the algorithms was in a single class. This pattern reduces the complexity and allows a single class for each algorithm. As chain of responsibility pattern encapsulates algorithms and the logic of selection, usage of complex switch statement is no more required. As each algorithm will be a subclass of CORStrategy, development or changes to algorithms is also uncomplicated.

## 7. Software Metrics

Software metrics [25] are used to provide programmers feedback about their program. Metrics can be used as guidelines to decide when to refactor. Software Metrics were used to analyze implementation of CalESStatement and CallGASStatement classes in PPCEA and PPCEA 1.0.

Eclipse provides a Metrics plug-in which gives information on different metrics. The following information taken adapted from [25] is about various metrics that can be analyzed in Eclipse.

1. Lines of Code (LOC): Total lines of code in the selected scope. Only counts non-blank and non-comment lines inside method bodies.
2. Number of Static Methods (NSM): Total number of static methods in the selected scope.
3. Number of Classes (NOC): Total number of classes in the selected scope
4. Number of Attributes (NOF): Total number of attributes in the selected scope.
5. Number of Packages (NOP): Total number of packages in the selected scope.
6. Method Lines of Code (MLOC): Total number of lines of code inside method bodies, excluding blank lines and comments.
7. Weighted Methods per Class (WMC): Sum of the McCabe Cyclomatic Complexity for all methods in a class.
8. Number of Overridden Methods (NORM): Total number of methods in the selected scope that are overridden from an ancestor class.
9. Nested Block Depth (NBD): The depth of nested blocks of code.
10. Number of Methods (NOM): Total number of methods defined in the selected scope.

11. Lack of Cohesion of Methods (LCOM): A measure for the Cohesiveness of a class. Calculated with the Henderson-Sellers method: If  $m(A)$  is the number of methods accessing an attribute  $A$ , calculate the average of  $m(A)$  for all attributes, subtract the number of methods  $m$  and divide the result by  $(1-m)$ . A low value indicates a cohesive class and a value close to 1 indicates a lack of cohesion and suggests the class might better be split into a number of (sub)classes.
12. McCabe Cyclomatic Complexity (VG): Counts the number of flows through a piece of code. Each time a branch occurs (if, for, while, do, case, catch and ternary operator, as well as the  $\&\&$  and  $\|\|$  conditional logic operators in expressions) this metric is incremented by one. Calculated for methods only.
13. Number of Parameters (PAR): Total number of parameters in the selected scope.
14. Abstractness (RMA): The number of abstract classes (and interfaces) divided by the total number of types in a package.
15. Number of Interfaces (NOI): Total number of interfaces in the selected scope.
16. Efferent Coupling (CE): The number of classes inside a package that depend on classes outside the package.
17. Number of Children (NSC): Total number of direct subclasses of a class.
18. Depth of Inheritance Tree (DIT): Distance from class Object in inheritance hierarchy.

The following information adapted from [25] shows safe ranges of some metrics.

1. Lines of Code (Method Level): Max 50 - If a method is over 50 lines of code it is suggested that the method be broken up for readability and maintainability.

2. Nested Block Depth (Method Level): Max 5 - If a block of code has over 5 nested loops, break up the method.
3. Lines of Code (Class Level): Max 750 - If classes have over 750 lines of code split up the class and delegate its responsibilities.
4. McCabe Cyclomatic Complexity (Method Level): Max 10 - If a method has over 10 different loops, break up the method.
5. Number of Parameters (Method Level): Max 5 - A method should have no more than 5 parameters. If it does, create an object and pass the object to the method.

Figures 7-1 and 7-2 show the metrics obtained for implementation of CallGASStatement in PPCEA and PPCEA 1.0, respectively.

Metric	Total	Mean	Std. Dev.	Maximum
Number of Overridden Methods	0			
Number of Attributes	1			
Number of Children	0			
▶ Method Lines of Code (avg/max per method)	938	67	122.46	476
Number of Methods	14			
▶ Nested Block Depth (avg/max per method)		2.857	1.457	6
Depth of Inheritance Tree	2			
▶ McCabe Cyclomatic Complexity (avg/max per		13.643	24.04	94
▶ Number of Parameters (avg/max per method)		1.786	1.081	4
Lack of Cohesion of Methods	0			
Number of Static Methods	0			
Specialization Index	0			
Weighted methods per Class	191			
Number of Static Attributes	0			

Figure 7-1 Metrics of CallGASStatement in PPCEA.

Metric	Total	Mean	Std. Dev.	Maximum
Number of Overridden Methods	0			
Number of Attributes	1			
Number of Children	0			
▶ Method Lines of Code (avg/max per method)	92	13.143	13.357	42
Number of Methods	7			
▶ Nested Block Depth (avg/max per method)		2	0.926	4
Depth of Inheritance Tree	2			
▶ McCabe Cyclomatic Complexity (avg/max per		2.857	1.641	6
▶ Number of Parameters (avg/max per method)		1	0.535	2
Lack of Cohesion of Methods	0			
Number of Static Methods	0			
Specialization Index	0			
Weighted methods per Class	20			
Number of Static Attributes	0			

Figure 7-2 Metrics of CallGASStatement in PPCEA 1.0.

As obtained in the results the mean of cyclomatic complexity was higher (13.643) for CallGASStatement in PPCEA and significantly reduced in PPCEA 1.0 to 2.857. Similarly the mean of block depth in PPCEA 1.0 reduced to 2 from 2.857. Because of modularization on PPCEA 1.0 weighted methods per class decreased to 20 from 191, the total number of lines of code reduced significantly from 938 to 92 which makes the classes easily readable and maintainable.

Figures 7-3 and 7-4 show the metrics obtained for implementation of CallIESStatement in PPCEA and PPCEA 1.0, respectively.

Metric	Total	Mean	Std. Dev.	Maximum
Number of Overridden Methods	0			
Number of Attributes	0			
Number of Children	0			
Method Lines of Code (avg/max per method)	652	54.333	63.535	239
Number of Methods	12			
Nested Block Depth (avg/max per method)		2.75	1.164	4
Depth of Inheritance Tree	2			
McCabe Cyclomatic Complexity (avg/max per		11.833	13.662	46
Number of Parameters (avg/max per method)		1.917	1.256	4
Lack of Cohesion of Methods	0			
Number of Static Methods	0			
Specialization Index	0			
Weighted methods per Class	142			
Number of Static Attributes	0			

Figure 7-3 Metrics of CalLESStatement in PPCEA.

Metric	Total	Mean	Std. Dev.	Maximum
Number of Overridden Methods	0			
Number of Attributes	0			
Number of Children	0			
Method Lines of Code (avg/max per method)	65	13	15.258	42
Number of Methods	5			
Nested Block Depth (avg/max per method)		1.6	0.49	2
Depth of Inheritance Tree	2			
McCabe Cyclomatic Complexity (avg/max per		2.4	1.2	4
Number of Parameters (avg/max per method)		0.8	0.4	1
Lack of Cohesion of Methods	0			
Number of Static Methods	0			
Specialization Index	0			
Weighted methods per Class	12			
Number of Static Attributes	0			

Figure 7-4 Metrics of CalLESStatement in PPCEA 1.0.

As obtained in the results the mean of cyclomatic complexity was higher (11.833) for CalLESStatement in PPCEA and significantly reduced in PPCEA 1.0 to 2.4. Similarly the mean of block depth in PPCEA 1.0 reduced 2.4 from 2.75. Because of modularization on PPCEA 1.0 weighted methods per class decreased to 12 from 142.

The results obtained prove that complexity due to conditional statements in CalIES and CallGAStatements in PPCEA 1.0 was greatly reduced compared to PPCEA. And because of modularization of components in CalIES and CallGA classes have fewer lines of code which make them more comprehensible.

## 8. Graphical User Interface for the PPCEA

The GUI for the PPCEA was designed using the Java Foundation Classes (JFC) [4]. JFC, which encompasses a group of features for building graphical user interface (GUIs) and adding rich graphics functionality and interactivity to Java applications [4]. The swing GUI components include all basic components while designing buttons, panes, and tables, among others.

Figure 8-1 shows the GUI that was designed for PPCEA 1.0, the users can input or run the program, view the result file, and export result files to Excel with a single click.

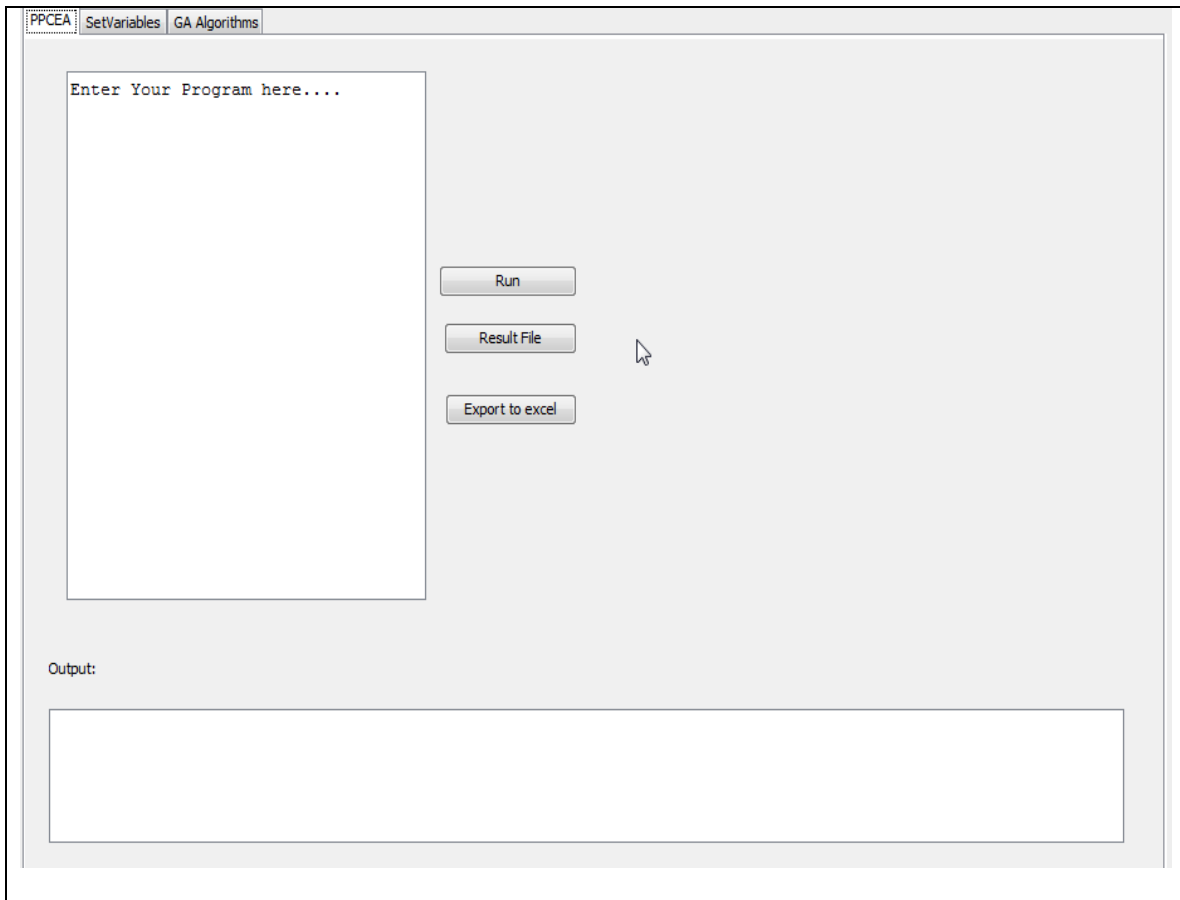


Figure 8-1 GUI for PPCEA 1.0.

The GUI for PPCEA 1.0 also lets end users to select different entropy/crossover/select parents/mutate algorithms easily as can be seen in Figure 8-2.

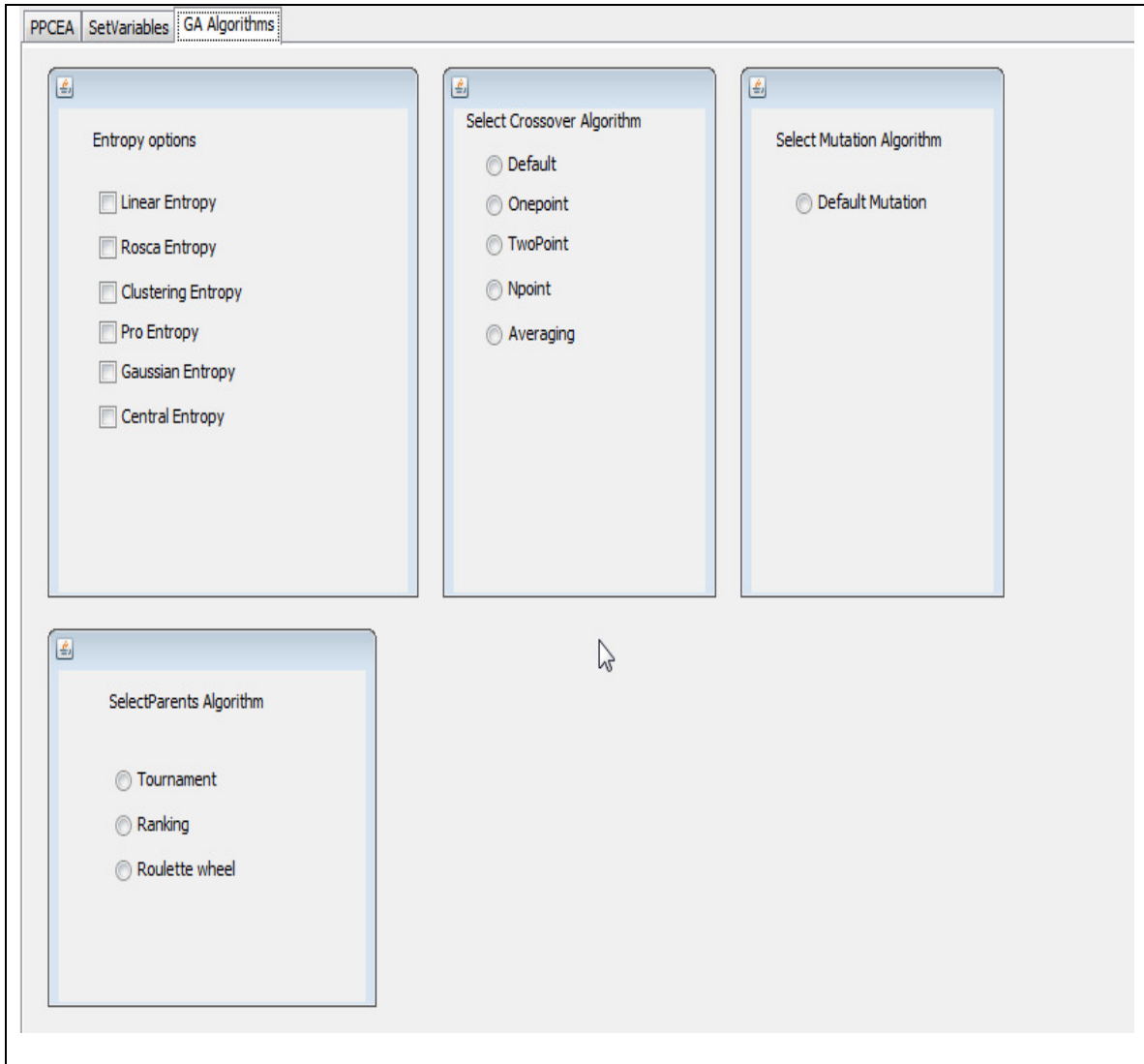


Figure 8-2 GUI for PPCEA 1.0.

With such GUI features users will be able to effectively use PPCEA 1.0.

## 9. Related Work

A large number of software tools have been developed in the last few years in the field of EAs (e.g., [26], [27] and [28]). GAFrame is a flexible and extensible framework for constructing GAs [26]. Evolutionary computing software (Paladin-DEC) enhances the concurrent processing and performance of evolutionary algorithms by allowing intercommunications of subpopulations [27]. Lil-gp is a C language system for developing genetic programming applications [28]. Previously mentioned systems do not deal with experimental studies in evolutionary algorithms. PPCEA is a powerful DSL which deals with experimental studies and also solves the problem of parameter control settings in EAs.

A number of researchers used design patterns for designing high quality software in diverse fields (e.g., control system [11], state search research application [12], CORBA systems [13]). One similar approach of using design patterns in Evolutionary Algorithms domain is JCLEC [14], a Java software system for the development of evolutionary computation applications. The design of Evolutionary Computation (EC) framework was greatly improved using design patterns [14]. Singleton, Abstract factory, Factory method, Builder, Prototype, Flyweight, Strategy, Visitor and Template method patterns have been used in the framework [14].

PPCEA applies the advantages of domain-specific languages to solve parameter setting problems. PPCEA not only outperforms the trial-and-error approach, but also performs the adaptable, reusable and controllable solutions of control parameter settings for EAs [1]. This aspect of PPCEA differentiates itself from any other software in EA

domain. By using design patterns in PPCEA 1.0, the semantics implementation of ES and GA is more enhanced and offers more advantages compared to PPCEA.

## 10. Conclusion

The main objective of this project was to make the design of EA in PPCEA much better (i.e., more modularized) so that any modification and development in the future is uncomplicated. By using Software Engineering principles and object oriented techniques those tasks are accomplished in the new design PPCEA 1.0. The results obtained from Eclipse software metrics tool have shown that complexity is greatly reduced in PPCEA 1.0.

The PPCEA 1.0 implemented various design patterns and has a number of advantages over PPCEA. The implementation of Visitor pattern makes PPCEA 1.0 easier to add or modify operations. With Strategy and Chain of Responsibility patterns, end users can now dynamically select different operations in EAs. Decorator pattern implementation resulted in reducing the execution time of PPCEA 1.0 programs, in scenarios where all entropy calculations are not required. Also, most of the EA components in PPCEA 1.0 are loosely coupled and independent development of those components is easier, also they are reusable.

The design of the GUI has made the PPCEA 1.0 easier to use. End users are provided an easy to use interface, and an important aspect of GUI is to allow them to select various algorithms easily.

Use of design patterns and software engineering principles made the structure of Evolutionary Algorithms in PPCEA 1.0 much comprehensible than PPCEA. Now the programmers can easily understand the structure, and development of evolutionary algorithms will be faster and uncomplicated.

## Bibliography

- [1] Shih-Hsi Liu, Marjan Mernik, Barrett R. Bryant, “*Parameter Control in Evolutionary Algorithms by Domain Specific Scripting Language PPCea*”, 1<sup>st</sup> Intl. Conf, on Bioinspired Optimization Methods and their Applications, pp. 41-50, 2004.
- [2] Zbigniew Michalewicz, “*Genetic Algorithms + Data Structures = Evolution programs*”, 3<sup>rd</sup> edition, Springer-Verlag, pp. 1-30, 1996.
- [3] Elisabeth Freeman, Kathy Sierra, Bert Bates, “*Head First design patterns*” O’Reilly Media, pp. 7-80, 2004.
- [4] <http://java.sun.com/docs/books/tutorial/uiswing/start/about.html>
- [5] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, “*Design Pattern*”, Addison-Wesley Publishing Company, pp. 1-70, 1995.
- [6] C. Alexander, S. Ishikawa, M. Silverstein, M. Jacobson, I. Fiksdahl-King, S. Angel, “*A Pattern Language*”, Oxford University Press, New York, pp 1-2, 1977.
- [7] Marjan Mernik, Jan Heering, Anthony M. Sloane, “*When and How to Develop Domain-Specific Languages*”, ACM Computing Surveys, Vol. 37, No. 4, pp. 316-344, 2005.
- [8] Robert C. Martin, “*The open-closed principle*”, In *More C++ gems*, pp. 97–112, 2000.
- [9] Robert C. Martin, “*The Liskov Substitution Principle*”, C++ Report, pp. 1-11, March 1996.
- [10] Robert C. Martin, “*The Dependency Inversion Principle*”, C++ Report, pp. 1-10, May 1996.

- [11] P. Dagermo, J. Knutsson, “*Development of an objectoriented framework for vessel control systems*”, Technical report, ESPRIT 111 I ESSI / DOVER #10496, Dover Consortium, pp. 1-2, 1996.
- [12] P. W. L. Fong, E. Kim, Q. Yang, “*Applying design patterns to state-space search applications*”, In 3rd Conference on the Pattern Languages of Programs (PLoP’96), Monticello (Illinois), pp. 1-2, 1996.
- [13] C. McKelvey, “*Design patterns achieve reuse in corba systems*”, In 3rd Conference on the Pattern Languages of Programs (PLoP ’96), Monticello (Illinois), pp. 1-2, 1996.
- [14] Sebastian Ventura, Cristobal Romero, Amelia Zafra, Jose A. Delgado, Cesar Hervas, “*JCLEC: a Java framework for evolutionary computation*”, *Soft Computing - A Fusion of Foundations, Methodologies and Applications*, Vol. 12, No. 4, pp. 381-392, October 2007.
- [15] F.P. Brooks Jr., “*No Silver Bullet: Essence and Accidents of Software Engineering*”, *Computer*, Vol. 20, No. 4, pp. 10–11, 1987.
- [16] David L. Parnas, “*On the Criteria to be used in decomposing System in to modules*”, in *Classics in Software Engineering*, New York, NY, Yourdon Press, pp. 139-150, 1979.
- [17] G. Yang, A. Sangiovanni Vincentelli, Y. Watanabe, F. Balarin, “*Seperation of Concerns: Overhead in Modeling and Efficient Simulation techniques*”, *Proceedings of the 4th ACM International Conference on Embedded Software*, pp. 44-53, 2004.
- [18] Shih-Hsi Liu, Marjan Mernik, Barrett R. Bryant “*To explore or to exploit: An entropy-driven approach for evolutionary algorithms*”, *International Journal of Knowledge-based and Intelligent Systems*, 13, pp. 1-14, 2009.
- [19] <http://www.silversoft.net/docs/dp/hires/pat3e.htm>

- [20] <http://userpages.umbc.edu/~tarr/dp/lectures/Chain-2pp.pdf>
- [21] Shih-Hsi Liu “*QoSPL : A quality of service-Driven Software Product Line Engineering Framework for Design and Analysis of Component-Based Distributed Real-Time and Embedded systems*” Ph.D. thesis, Dept. of Computer and Information Sciences, University of Alabama at Birmingham, pp.101-233, 2007.
- [22] [http://students.cis.uab.edu/liush/PPCea\\_files/page0001.htm](http://students.cis.uab.edu/liush/PPCea_files/page0001.htm)
- [23] Matjaz B. Juric, Benny Mathew, Poornachandra Sarang, “*Business Process Execution Language for Web Services*”, Packet Publishing, pp. 5-7, 2006.
- [24] James R. Groff, Paul N. Weinberg, “*SQL, the complete reference*”, McGraw Hill Osborne, pp 1-15, 2000.
- [25] <http://agile.csc.ncsu.edu/SEMaterials/tutorials/metrics/>.
- [26] Angela S. Chuang, Felix Wu, “*An Extensible Genetic Algorithm Framework for Problem solving in a Common environment*”, IEEE Transactions on power systems Vol.15, No. 1, pp. 269-275, 2000.
- [27] K. C. Tan, A. Tay, J. Cai, “*Design and implementation of a distributed evolutionary computing software*” IEEE Trans Syst Man Cybern Part B, pp 325-338, 2003.
- [28] <http://garage.cse.msu.edu/software/lil-gp/>.