

A GUI Driven Platform for
Implementing Evolutionary Algorithms in Java

Reza Etemadi

A Thesis
in
The Department
of
Electrical and Computer Engineering (ECE)

Presented in Partial Fulfilment of the Requirements
for the Degree of Master of Applied Science at
Concordia University
Montreal, Quebec, Canada

August 2014

© Reza Etemadi, 2014

**CONCORDIA UNIVERSITY
SCHOOL OF GRADUATE STUDIES**

This is to certify that the thesis prepared

By: Reza Etemadi

Entitled: “A GUI Driven Platform for Implementing Evolutionary Algorithms in Java”

and submitted in partial fulfillment of the requirements for the degree of

Master of Applied Science

Complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the final examining committee:

_____	Chair
Dr. M. Z. Kabir	
_____	Examiner, External
Dr. O. Ormandjieva (CSE)	To the Program
_____	Examiner
Dr. A. Agarwal	
_____	Examiner
Dr. A. Hamou-Lhadj	
_____	Supervisor
Dr. N. Kharma	
_____	Supervisor
Dr. P. Grogono (CSE)	

Approved by: _____
Dr. W. E. Lynch, Chair
Department of Electrical and Computer Engineering

_____20_____

Dr. Amir Asif, Dean
Faculty of Engineering and Computer Science

Abstract

A GUI Driven Platform for Implementing Evolutionary Algorithms in Java

Reza Etemadi

CodeMonkey-GA (CM) is a GUI driven software development platform that enables non-experts and experts alike to turn an evolutionary algorithm design into a working Java program, with a minimum amount of manual coding. CM is provided as a framework and plug-in application for the Eclipse platform for non-commercial uses.

We compare some of the most popular frameworks and platforms for evolutionary computation. We discuss their shortfalls and justify the need for still another platform. Hence, we present CodeMonkey-GA: its concept, internal architecture and design. We provide an overview of the graphical user interface (GUI) of the platform followed by examples of evolutionary algorithm applications, all generated using CodeMonkey's Eclipse application.

Through several examples we demonstrate the ease of use and (to some degree) the applicability of the CM application. The Ackley function is a well-known test function for optimization; the Traveling Salesman Problem is a famous example of NP-Complete problems; the Knapsack problem is an example of combinatorial optimization. In all three cases, CM is used to develop working Java programs that provided satisfactory solutions, which are as good as, or better than the given solutions. Critically, in all cases, not a line of code was entered or altered – bar the fitness function – by the user.

Acknowledgement

I am very thankful to my supervisor Dr. Nawwaf Kharma for working closely with me in shaping the scope of the thesis and providing invaluable feedback throughout the specification, implementation and documentation phases of this work.

I would like to thank my co-supervisor Dr. Peter Grogono for his insights and contributions to the papers that reported on our work.

I would also like to thank my family for supporting me throughout my academic journey.

Table of Contents

1	Introduction & Review	1
1.1	Competition	1
1.2	Comparison Table	5
1.3	Motivation.....	6
2	Design & Implementation	7
2.1	Concept	7
2.2	The Framework	7
2.2.1	The Core	8
2.2.2	Packaging & Structure.....	13
2.3	Plug-in Application	14
2.3.1	Genotype Representation.....	16
2.3.2	Population Initialization	17
2.3.3	Fitness Calculation	18
2.3.4	Termination Criteria.....	20
2.3.5	Parent Selection	21
2.3.6	Variation Operations.....	22
2.3.7	Survivor Selection	24
2.4	Program Execution.....	25
2.5	Use of Eclipse JDT and AST.....	26
3	Examples of Application.....	28
3.1	The Ackley Function	28
3.1.1	Problem Description	28
3.1.2	Solution Outline	28
3.1.3	Implementation Details	28
3.1.4	Result	36
3.2	The Traveling Salesman Problem.....	36
3.2.1	Problem Definition.....	37
3.2.2	Implementation	37
3.2.3	Result	38

3.3	The Knapsack Problem	39
3.3.1	Problem Definition	39
3.3.2	Implementation	40
3.3.3	Result	42
3.4	Image Noise Cancelation.....	43
3.4.1	Problem Definition	43
3.4.2	Implementation	44
3.4.3	Result	51
4	Summary & Conclusion.....	55
5	Future Work	56
	Appendix A.....	60
	Appendix B.....	63

List of Figures

Figure 1. General flow of evolutionary algorithms	7
Figure 2. Class diagram	8
Figure 3. Sequence diagram.....	12
Figure 4. Use case diagram	14
Figure 5. Activity diagram of CM plug-in application.....	15
Figure 6. Genotype definition	16
Figure 7. Population initialization	18
Figure 8. Fitness calculation.....	19
Figure 9. Termination strategy.....	21
Figure 10. The customizable generic parent selection mechanism adopted by CM	22
Figure 11. Variation operations	24
Figure 12. Survivor selection in CM	25
Figure 13. Activity diagram of code execution	25
Figure 14. Genotype definition step	29
Figure 15. Population initialization step	29
Figure 16. Fitness calculation step.....	30
Figure 17. Termination criteria step	31
Figure 18. Parent selection step	32
Figure 19. Variation operations step	33
Figure 20. Survivors selection step	34
Figure 21. The progress in fitness over generations in Ackley's problem	36
Figure 22. Progress in fitness over generations in TSP	39
Figure 23. Progress in fitness over generations in knapsack problem.....	42
Figure 24. ImageJ salt and pepper noise.....	43
Figure 25. Sample image from Berkley data set, before and after salt & pepper noise	44
Figure 26. Remove outlier filter input parameters.....	50
Figure 27. Progress of fitness over the course of the best run of image denoising	51
Figure A.28. CM icon in Eclipse toolbar	60
Figure A.29. Import project wizard source selection.....	61
Figure A.30. Import project wizard.....	61
Figure A.31. Eclipse environment after CM installation	62
Figure B.32. CM initial page	63
Figure B.33. Configuration wizard.....	64
Figure B.34. Genotype definition wizard	65
Figure B.35. Homogeneous genotype.....	66
Figure B.36. Variation operators.....	66
Figure B.37. Heterogeneous genotype subpart selection	67
Figure B.38. Heterogeneous genotype wizard.....	68
Figure B.39. Selecting homogeneous of heterogeneous option	68
Figure B.40. Homogeneous of heterogeneous genotype definition	69

Figure B.41. Homogeneous of heterogeneous fixed length	70
Figure B.42. Homogeneous of heterogeneous variable length	70
Figure B.43. Population initialization wizard	71
Figure B.44. Fitness calculation wizard	72
Figure B.45. Fitness calculation wizard detail	73
Figure B.46. Phenotype source file for fitness code entry	74
Figure B.47. Termination criteria wizard	75
Figure B.48. Resource exhausted condition	75
Figure B.49. Parent selection wizard	76
Figure B.50. Fitness transformation option	77
Figure B.51. Linear fitness transformation option	77
Figure B.52. Logarithmic fitness transformation option	77
Figure B.53. Power law fitness transformation option	78
Figure B.54. Variation operations wizard	79
Figure B.55. Variation operation branches definition	80
Figure B.56. Survival selection wizard	81
Figure B.57. Survival selection options	81
Figure B.58. Survival selection detail	82
Figure B.59. Survival selection elitism and re-initialization options	83
Figure B.60. CM entry point to generated code	84
Figure B.61. Running the generated code	85

List of Table

Table 1. Camparative summary of common EA platfroms	5
Table 2. List of parameters and their values for the Ackley problem.....	34
Table 3. List of parameters and their values for the TSP	37
Table 4. Unbounded knapsack criteria	40
Table 5. List of parameters and their values for knapsack problem.....	40
Table 6. - Unbounded knapsack best found solutions.....	43
Table 7. Parameters and their values used for the image noise cancelation problem	47
Table 8. Sample RMS and output of best, least and median found solutions by CM.....	52
Table 9. Summary of RMS comparison between CM and ImageJ filters	53
Table 10. Sample comparison of RMS and output images of CM and filters	53

1 Introduction & Review

Evolutionary Algorithms (EA) have broad applications not only in computer science but also in various fields of engineering, economics, finance, art and many branches of science such as physics, chemistry, biology and so on. Examples of application include Collision Avoidance Control [1], Industrial Design [2], Evolutionary Art [3], Pharmaceutical Drug Design [4].

Because of the broad range of application domains, the landscape of EA computational tools is rich with frameworks, libraries and platforms with various capabilities and based on different programming languages. Despite a wealth of options, there are gaps that justify the investment in a new EA software platform. To demonstrate these gaps, it is only right that we survey the best existing solutions and then describe how Code Monkey targets the shortcomings of the existing alternatives.

Designing a comprehensive tool that covers the different flavors of Evolutionary Algorithms and that is also easy to use is not a simple task. In this section we intend to look at the existing tools objectively, highlighting the areas that a new tool can cover.

1.1 Competition

There are various implementations of Evolutionary Algorithms (Genetic Algorithms or GA, Genetic Programming or GP, Evolutionary Strategies or ES, and Evolutionary Programming or EP) offered by different authors and vendors. These implementations differ from each other in:

- The scope they cover; some of them are application specific like “Object-Oriented Framework for Genetic Algorithms with Application to Space Truss Optimization” [5] and some are general libraries like the genetic algorithm package, which is part of Apache Foundation common math library [6].
- Their performance and scalability, which is also influenced by the programming language and platform they are coded in or deployed on, respectively.

- The learn ability and usability of the implementation, which may come in the form of a library, framework, application or platform.

In the sequel, we review the most widely used and referenced implementations, not specific to one domain of application.

GEATbx [7]. Genetic and Evolutionary Algorithm Toolbox for use with Matlab (GEATbx) is a Matlab tool and one of the few packages that cover the four main flavors of EA (GA, GP, ES and EP). It allows one to define a homogenous genomic representation using one of four primitive data types. It has limited support for heterogeneous genotypes [4]. It has no GUI for novice users. It is proprietary, and has limited number of selection and genetic operations (i.e., crossover and mutation). It is not extendable via 3rd party components and is currently sold under different levels of licensing.

EO [8]. Evolving Objects (EO) is a template-based C++ open-source library/framework for writing stochastic optimization programs [9]. It allows for homogenous genomic representations using most primitive types, except for integer. It has no out-of-the box support for heterogeneous genotypes and their variation operations. Many different selection mechanisms are provided. It does not have a GUI based entry or customization ability. It is open-source and free to use under GPLv2. It is extendable via 3rd party and some extensions such as PARADISEO (PARAllel and DIStributed Evolving Objects) [10] enable distributed implementations of the framework.

EASEA [11]. EAsy Specification of Evolutionary Algorithms (EASEA) is a parallel Artificial Evolution platform developed by a team at Université de Strasbourg. It covers many types of optimization problems (continuous, discrete, combinatorial, mixed and more) using Genetic Programming.

The main disadvantage of EASEA is the introduction of a special-purpose language that is not widely used. There is no built-in genotype for primitive types and also no support for heterogeneous genotypes. It is not easily extendable through 3rd party components because of its special language. It supports computation in distributed environments using DEARM. It is free to download and use under LGPL.

DREAM [12]. Distributed Resource Evolutionary Algorithms Machine (DREAM) project (funded by the European commission) seeks to provide the technology and software infrastructure necessary to support the next generation of evolving info-habitants in a way that makes that infrastructure universal, open and scalable.

Its software framework (Java Evolutionary Object) has many genotypes defined with many related operations. It does not provide any heterogeneous genotype. It has a GUI for non-programmers for both I/O interaction and problem definition. It is open source and offered under a GPL license. The platform has a distributed architecture and the framework has an API that facilitates distributed implementations.

Watchmaker [13]. Watchmaker Framework for Evolutionary Computation is a software framework for implementing evolutionary algorithms in Java. It is used in the Apache Mahout Project [14] and some specialized frameworks such as GEP4J (Gene Expression Programming for Java) [15].

It does not provide any predefined genotypes and relies on types defined in another package (uncommons.maths) that is not provided in the bundle. There is a GUI for monitoring progress but it does not have any GUI for non-experts to generate code. It only supports pipelining of operations. The number of variation operators provided out-of-the-box is very limited. It is open source and available under Apache software license and easily extendable.

JGAP [16]. Java Genetic Algorithms Package (JGAP) (pronounced "jay-gap") is a Genetic Algorithms and Genetic Programming component provided as a Java framework. It provides basic genetic mechanisms that can be used to implement evolutionary solutions to problems.

It has many out-of-the-box genotypes. There is no heterogeneous genotype support in the framework. Some of the limitations of the Watchmaker framework related to genetic operators are also present here. It does not provide any GUI for end-users to interact with to generate code; only expert users who must first learn the framework and has prior knowledge of GA or GP can make use of JGAP. It is however an open source project, free and extendable under LGPL and Mozilla licenses.

JCLEC [17]. Java Class Library for Evolutionary Computation (JCLEC) is a software system for EA written in Java. It is a high level framework that supports GA, GP and EP. Its architecture has three layers: the core that includes abstract definition, base implementations and utility classes; the second layer is the runner that executes the EA process based on a user defined configuration file; the last layer is GUI interface that allows a user to define the parameters for a configuration and view the result of execution.

JCLEC targets both expert and novice users and therefore it offers more options to users. However the framework does not have any built-in support for heterogeneous genotype. Also the fact that GUI generates XML file instead of code it does not easily allow experts users to adopt a hybrid approach of generating some of the code using the GUI then directly entering or editing other parts.

JCLEC does not provide a GUI based mechanism of selecting different combinations of variation operators. Another area of improvement is documentation and in particular the in-context help for the GUI application.

ECJ [18]. ECJ is a feature rich Evolutionary Computation framework written in Java. It covers many different implementations of EA. It is very flexible and it loads the algorithms dynamically at runtime based on user entered parameters. It is designed for large scale usage and supports multi-objective optimization, island models, co-evolution and many more features. The framework support fixed and variable length genotypes and there is wealth of variation operators. It has many pre-defined GP and GA application problem domains. The package includes a GUI for charting.

The framework heavily relies on parameter files to configure and instantiate all of the EA processes. However there is no GUI or wizard process to help users go over configuration in a stepwise manner, leading to the generation of the parameters file. This means that the user needs to get familiar with numerous parameter names and value ranges. This is an error prone method and any problem is not detected until runtime.

Because it was originated from a GP project, it is more suited for that branch of EA. Also the framework does not provide any heterogeneous genotype.

1.2 Comparison Table

The result of comparison is summarized in table 1. The review factors are listed at the first column of the table.

Table 1. Camparative summary of common EA platfroms

Name	GEATbx	EO	ESAEA	DREAM	Watchmaker	JGAP	JCLEC	ECJ
Programming Language	Matlab	C++	EASEA	EASEA /Java	Java	Java	Java	Java
Type	Library	Framework	Platfrom	Platform	Framework	Framework	Platform	Framework
Homogeneous Genotypes	Real, Integer, Binary, Permutation	Real, Binary, Permutation	Not defined	Real, Integer, Binary, Tree	Real, Byte, Integer, BitString (in a different lib)	String, Integer, Binary, Real & more	Real, Integer, Binary, Tree	Real, Integer, Binary, Tree
Heterogeneous Genotypes	None	None	None	None	None	None	None	None
Selection Types	4	9 (more pluggable)	Not documented	11 (more pluggable)	6 (more pluggable)	4 (more pluggable)	12 (more pluggable)	12 (more pluggable)
GUI for Novice End-users	Yes	No	No	Yes	No	No	Yes	Only for charting
Open Source	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Extendable	No	Yes	Limited	Limited	Yes	Yes	Yes	Yes
Distributed Environment Support	Possible by Matlab	Available (by extensions, ParadisEO)	Yes	Yes	Yes	Yes	Not defined	Yes
Licencing /Pricing	Multi-tier /Not-free	GPLv2 / Free	LGPL / Free	GPL / Free	Apache v2.0 / Free	LGPL and Mozilla / Free	GPL /Free	Academic Free Licence /Free

1.3 Motivation

The comparison table demonstrates that some EA platforms offer a GUI for end-users. In case of GEATbx it is bound by Matlab limitations and in case of DREAM and JCLEC it is a relatively out of date Java Swing based GUI; all other implementations require programmers able and willing to learn new frameworks. This is a major hurdle for end-users with limited background in computer programming. EA platform users from disciplines other than computer science and engineering would greatly benefit from a GUI based stepwise means of automatic code generation.

In addition, none of the reviewed EA platforms provide built-in support for heterogeneous genotypes (where genes are made up of different types). While defining heterogeneous genotypes is possible with most of the reviewed platforms, it is an asset to have the capability of constructing a heterogeneous genotype through the combination of two or more different homogeneous genotypes; if this comes with the complementary capability of choosing appropriate variation operations then so much the better.

Furthermore, other implementations excepting JGAP do not provide any mechanism for data serializing. This feature is quite handy if one is to invoke an external program for fitness calculation. This is important as real-world situations often require interacting EA software solutions with well-established - even proprietary - external software that is tried and tested by experts in the field of application.

In contrast, we provide an easy-to-learn and use GUI-driven platform for generating different flavors of EA and for a wide range of target applications. It supports both homogeneous and heterogeneous genotypes with appropriately defined variation operators. It also offers a good degree of flexibility in both offspring generation and survivor selection.

At the same time the framework part of the platform is open and can be directly used by expert users, who wish to customize the resulting evolutionary algorithm program in ways that are not possible via the GUI. A GUI driven platform with direct coding capabilities, in a widely adopted language, allow more experienced users quick code generation followed by custom modification, leading to an EA Java program that meets the exact needs of the programmer.

2 Design & Implementation

Code Monkey- GA (CM) is the result of an ongoing project that takes advantage of modern features of Java (e.g., Generics and Annotation), combining them with the power & ease of use of the Eclipse platform, to provide an EA development framework for expert users and an Eclipse plug-in application for novice users.

2.1 Concept

As Evolutionary Algorithms are population based, an EA may be visually represented (as in Figure 1) by three pools, with individuals transiting to & from them, during various stages of evolution; this is described via high-level pseudo-code as well, for those who prefer it.

BEGIN

INITIALIZE population with random candidate solutions;

EVALUATE each candidate;

REPEAT UNTIL (*TERMINATION CONDITION* is satisfied) **DO**

1 *SELECT* parents;

2 *RECOMBINE* pairs of parents;

3 *MUTATE* the resulting offspring;

4 *EVALUATE* new candidates;

5 *SELECT* individuals for the next generation;

OD

END

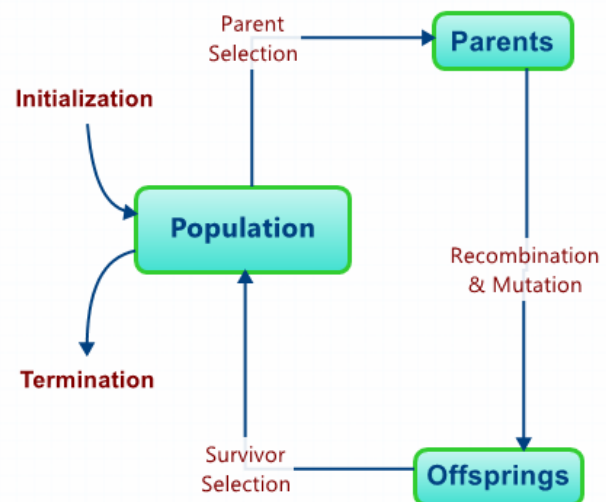


Figure 1. General flow of evolutionary algorithms

2.2 The Framework

The CM framework is provided as a zip file. It can be imported into Eclipse as a Java Project. To explore the framework, we start by discussing its architecture and data model which are based on object-oriented concepts and design patterns.

2.2.1 The Core

The class diagram (Figure 2) shows the core package of the framework. It includes the main components that we describe below.

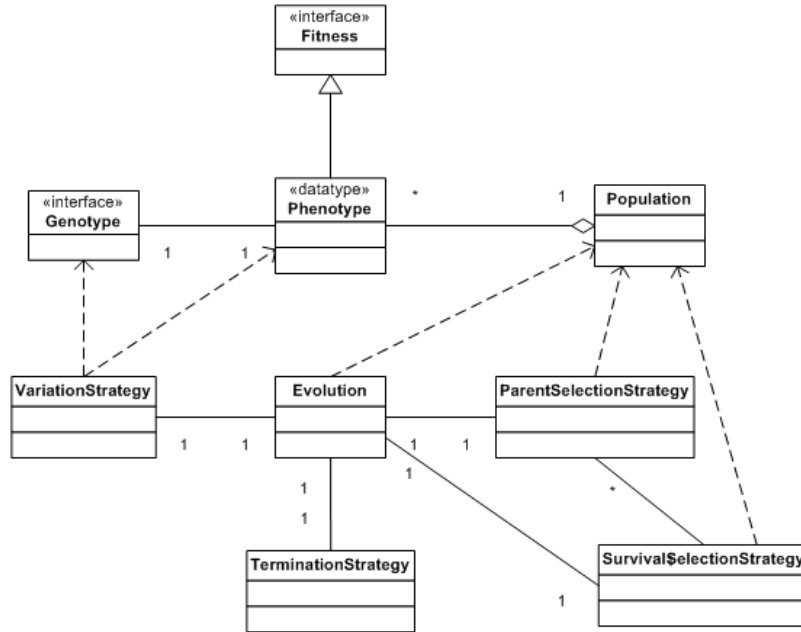


Figure 2. Class diagram

Phenotype: This class represents an individual solution. This is an abstract class that has two other elements, Genotype and Fitness. It also has an optional object called data that is only needed if an implementation needs to store and process any epi-genetic data related to phenotype (e.g. Epigenetic Algorithm [19]). Any implementation based on the framework, including those generated by the CM application, will create a concrete subclass of the Phenotype class for representation of their individual solutions. Phenotype uses Java generics for its Genotype and Fitness elements.

Genotype: This is the component that represents the encoding part of Phenotype in the search space. It is defined as a java interface that extends java Collection, Cloneable and Serializable interfaces. Being an interface allows any Java class to be used as a Genotype as long as it implements the methods of the interface. Since there are many existing implementations of the Java Collection interface (e.g. ArrayList, Vector etc.): they can be easily extended to represent Genotypes in the framework.

In fact the 'genotype' sub-package in the framework (edu.ccil.ec.genotype) extends Java ArrayList, LinkedList, TreeSet and Vector to provide an abstract level for genotype representation in the framework. Out of these abstract classes the ArrayListGenotype is further extended in the framework (edu.ccil.ec.genotype.arraylist) to provide implementation for data types such as Boolean, Integer, Permutation and Real values. These classes are used for building homogenous and heterogeneous genotypes in the CM application based on user input.

Homogenous in this context means all genes have the same type such as Boolean. Where heterogeneous means that genes are from different types. The only constrain is that a gene at a specific allele always has the same type and variation operations do not violate the organization of heterogeneous genotype.

Since variation operations are data type specific there is an equivalent operator class per genotype data type that provides several mutation and crossover operations related to that data type. These ready-to-use mutation and crossover operators are also used in the CM application to generate variation strategies based on user input. These operators will be described in the variation strategy section.

Fitness: The fitness component is used to represent how suitable a phenotype is as a solution in comparison to other phenotypes. This is also defined as a Java interface in the framework that extends Java Comparable interface. While, in most cases, Real values are used for representing fitness, this interface uses Java generics to provide more flexibility, by accepting any subclass of Java Number class that implements Comparable interface.

Population: This class acts as the container of Phenotypes. It is a concrete class with generics that accepts any subtype of Phenotype class. This class (or its subtypes) can be registered to represent the initial population, parent pool, offspring and next generation. This class has many utility methods for random picking, sorting, getting different statistics of the population, based on fitness of individuals and so on.

TerminationStrategy: This class represents the termination criteria in the framework. All possible criteria that are provided for the CM application are defined in this class, but the class is abstract. Once a subclass that implements the abstract method 'check()' is registered in the framework, it will be invoked iteratively to check whether the process should stop or continue.

ParentSelectionStrategy: This is the class that applies parent selection based on mechanisms described below. It has one abstract method that represents the selection process inside the selection window. Three subclasses of this class for Truncation, Proportional (Roulette wheel) and Random selection are provided by the framework. Any concrete subclass of ParentSelectionStrategy that is registered will be invoked to create the parent pool.

VariationStrategy: This is an abstract class that represents the mating process in the framework. A concrete subclass of this class will invoke the variation methods that are implemented in the Genotype based on the arity (e.g., unary or binary) of the method with the desired probability and sequence. The class must be registered in order to be invoked to create an offspring pool from the parent pool.

As mentioned earlier the framework provides a set of variation operators for ArrayList Genotypes based on Boolean, Integer, Permutation and Real data types. Here is a complete list of these operators:

Boolean Operators: One Point crossover, Two Point crossover, Uniform crossover, Flip Mutation, One-Position Mutation.

Integer Operators: One Point Crossover, Two Point Crossover, Discrete Recombination, One-Position Mutation and Creep Mutation.

Permutation Operators: PMX Crossover, Order Crossover, Cycle Crossover, Swap Mutation, Insert Mutation, Inversion Mutation.

Real Operators: Discrete Recombination, Continuous Recombination, Convex Recombination, Local Crossover, One-Position Mutation, Creep Mutation.

For heterogeneous genotypes there are three crossover operators applicable to heterogeneous genotypes. They are: One Point, Two Point and Uniform Crossover. They are independent of gene type.

When building a variation strategy, any arrangement of these operators, with different probabilities, can be combined. In fact, the CM application uses these operators to provide end users with a list of available operators based on selected genotypes. This allows users to define

the variation strategy by selecting operators and entering probabilities. This is described in more detail in the CM application section.

SurvivalSelectionStrategy: This abstract class represent the survivor selection process in the framework. It internally relies on ParentSelectionStrategy as described before. A concrete subclass will need to define what percentage of the next population comes from available populations and based on what selection mechanisms. The subclass need to be registered to be invoked to create the next generation during the process.

FitnessTransformer: Fitness transformation is optionally used in any selection implementation to scale or transform the fitness before the selection is done. FitnessTransformer is an abstract class that represents this need and certain concrete subclasses are provided in the framework for linear, exponential and ranking scaling. If a subclass of this class is registered or set in the ParentSelectionStrategy it will be invoked before the selection is applied inside each window.

ESException: This is a general class that represents any runtime exception related to the evolutionary process. Several error codes are pre-defined and used throughout the framework to identify the cause of errors that might occur in the process.

Evolution: This class is the orchestrator of the evolutionary process in the framework. It is an abstract class that implements the general logic of the evolutionary process. Any implementation based on the framework will create a concrete subclass of this class and include a main method so it can be invoked as a Java application.

All above mentioned registrations of data types and strategies need to be defined in the registration() method of a concrete subclass of Evolution class. Once all necessary elements are registered the class can act as the starting point of execution of the evolutionary process. The Evolution class uses Factory pattern and class registration is done using reflection. The following code is a sample of this registration procedure.

```

@Override
public void registration(){
    //Any registration of strategy or type that is used in the process goes here
    register(Phenotype.class,new Phenome());
    register(Population.class, new PhenomePopulation());
    register(ParentSelectionStrategy.class, new edu.ccil.ec.selection.Proportional(20, 15, true));
    register(VariationStrategy.class, new myVariationStrategy(true));
    register(TerminationStrategy.class, new myTerminationStrategy());
    register(SurvivalSelectionStrategy.class, new mySurvivalSelectionStrategy());
    register(RandomEngine.class, new RandomEngine());
    register(FitnessTransformer.class,new edu.ccil.ec.selection.transform.Rank());
    return;
}

```

Figure 3 shows how and in what sequence the Evolution class invokes the registered strategies over the course of one generation.

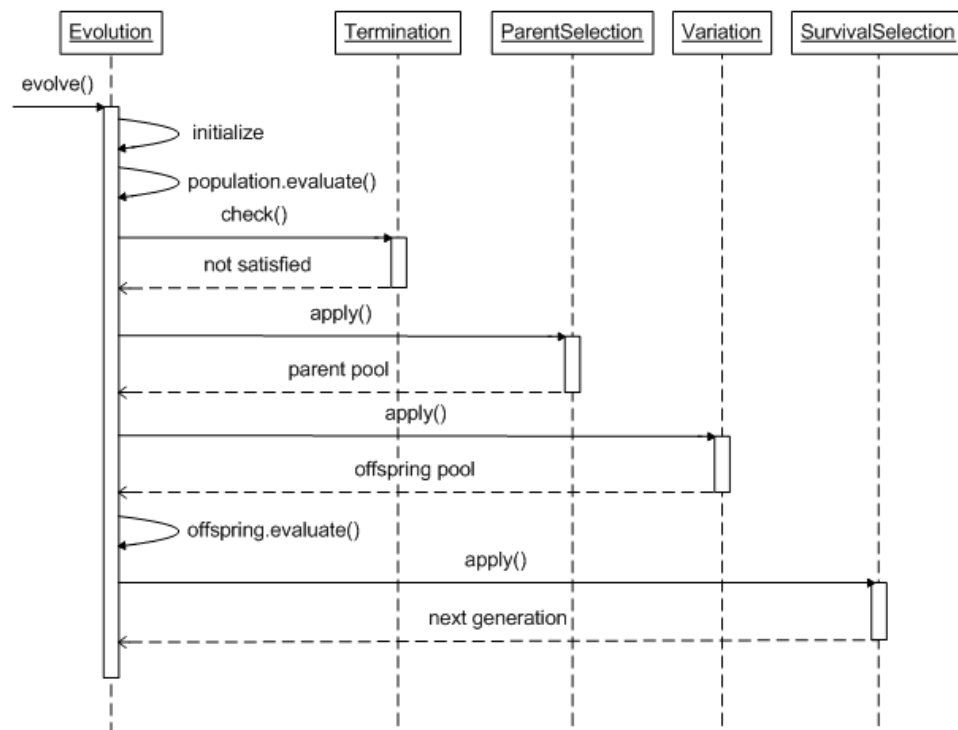


Figure 3. Sequence diagram

Once the `evolve()` method is called, it starts with initialization, which includes registration of all strategy classes and initializing the first generation. Then the first generation is evaluated. Hence, Evolution calls the `check()` method of the `TerminationStrategy` class. If the termination criteria is not satisfied then `apply()` method of `ParentSelectionStrategy` class is called to create the parent pool. Hence, the `apply()` method of the `VariationStrategy()` class is called to generate the offspring pool from the parent pool. Then, the offspring pool is evaluated and the `apply()` method of `SurvivalSelectionStrategy` class is called to create the next generation. The next generation replaces the current generation. This cycle is repeated until the `check()` method in `TerminationStrategy` class returns true. Once that happens, the evolutionary process stops and the best available individual is returned.

2.2.2 Packaging & Structure

The core of framework that contains all interfaces and abstract classes of all above mentioned components is in the **edu.ccil.ec** package. There are several sub packages in the framework that we describe here:

edu.ccil.ec.genotype: This package include abstract classes that extend some classes in the Java Collection API with implementation of genotype interface. Examples are: `ArrayListGenotype`, `LinkedListGenotype`.

edu.ccil.ec.genotype.arraylist: This package provides concrete implementations of `ArrayListGenotype` for primitive types. For each genotype there is also an `Operator` class that includes all mutation and crossover operations related to that specific genotype.

edu.ccil.ec.selection: This package contains the three types of selection, deterministic probabilistic and random, which are part of the selection strategy architecture in CM.

edu.ccil.ec.selection.transform: This package includes all different fitness transformation schemes that are provided by CM. They are all implementations of the `FitnessTransformer` interface in the core package.

edu.ccil.ec.tool: This package contains internal utility classes that are used by the rest of the package (e.g., example logging, random generator).

edu.ccil.ec.plugin_template: This package contains the base classes used by the CM Plug-in Application to generate code based on input provided by user.

In addition to the above mentioned packages there are two other top level packages. One is example that contains many sub packages. Each is an example generated using the plug-in application for known problems (e.g., Traveling Salesman Problem, several multi-dimensional multi-modal mathematical functions). It also contains one advanced example of Image possessing.

The other package is called external which contains classes that are used for fitness calculation of the example package.

2.3 Plug-in Application

CodeMonkey application is built on top of the Eclipse platform. It uses Eclipse's plug-in architecture [20] to create a GUI-based application. The CM plug-in application allows end-users to provide customizing input reflecting a specific EA flavor and target application. The CM application uses the Eclipse JDT (Java Development Tools) API to create the necessary code, which runs on the CM framework. The user can then launch the generated Java program in Eclipse or independently of it.

The following use case diagram demonstrates the two types of interactions between a user and the CM.

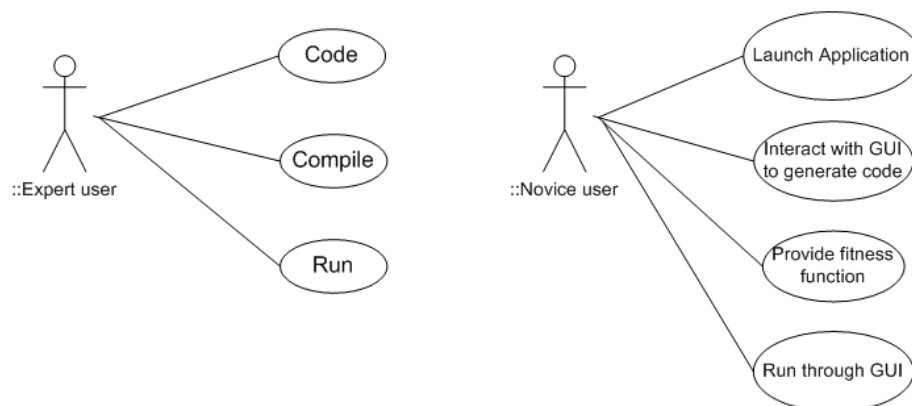


Figure 4. Use case diagram

As shown, two types of users can apply CM to customize and generate an EA in Java: novice and expert. Expert users can directly work with the framework by using existing functionalities or extending them and adding new implementations. Novice users are asked to provide the CM application with customizing input, and this allows CM to generate a Java program that implements a specific EA flavor for a specific EA application. The only part of interaction with CM application that necessitates the provision of either (1) actual code or (2) a link to an external program or function is the fitness function. This is justified as the fitness function is that part of EA process that pertains to the specific problem, which could come from any domain. It is also noteworthy that one user can be involved in both types of use cases: doing part of the work via the CM application and modifying or/and adding code to the resulting Java program, manually.

Figure 5 exhibits the order of user interactions between a user and the CM application, to generate and run an EA application.

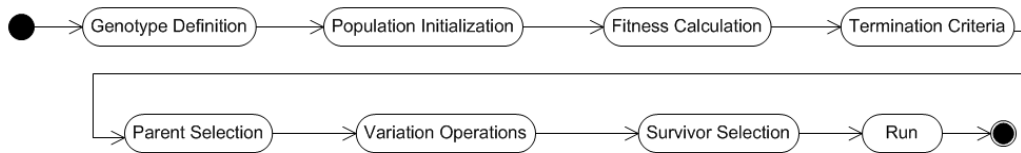


Figure 5. Activity diagram of CM plug-in application

As shown, once the CM plug-in application is launched, the first step is defining the genotype structure, followed by defining how the population will be initialized. Hence, the user defines how fitness will be calculated. This is the only step that necessitates manual code entry or external communications. The next step is defining what the termination criteria are. This is followed by defining the mechanism for parent selection. The remaining two steps are defining how variation operations are applied and how the next generation is created. Once those steps are completed, code generation is completed and the code can be compiled and executed.

In the example section we will demonstrate each of the above steps in detail and show using screenshots how a user interacts with the CM plug-in application to generate and run a sample EA program. But here we describe the concept and internal design behind each step.

In the sections to follow, we describe in detail, how the application interacts with user to deliver the end result. Once the Eclipse plug-in is installed, the CM icon will appear in the Eclipse toolbar. Clicking on the icon will load the CM cheat sheet that guides the user throughout the development process. Every step in the development process has a clear help page associated with it and the wizard that takes one through that process is constantly verifying user input and hinting at typical valid entry values.

2.3.1 Genotype Representation

Having the right genotype representation has significant impact on the success of an evolutionary algorithm. CM divides genotype representation into three main categories: homogenous, heterogeneous and homogeneous of heterogeneous. A homogenous genotype is a collection of same primary type genes (e.g. Boolean, Integer, Real or Permutation). Heterogeneous genotype is a collection of homogeneous genes of different primary types. CM supports basic homogenous genotypes as part of the framework and easily allows building heterogeneous genotypes by combining homogenous types using the application GUI. To provide further flexibility, CM allows the end-user to utilize the heterogeneous genotype as a gene for creating a homogeneous genotype. Essentially allowing the creation of homogeneous genotypes with custom made genes. At this level the length of genotype can be fixed or variable.

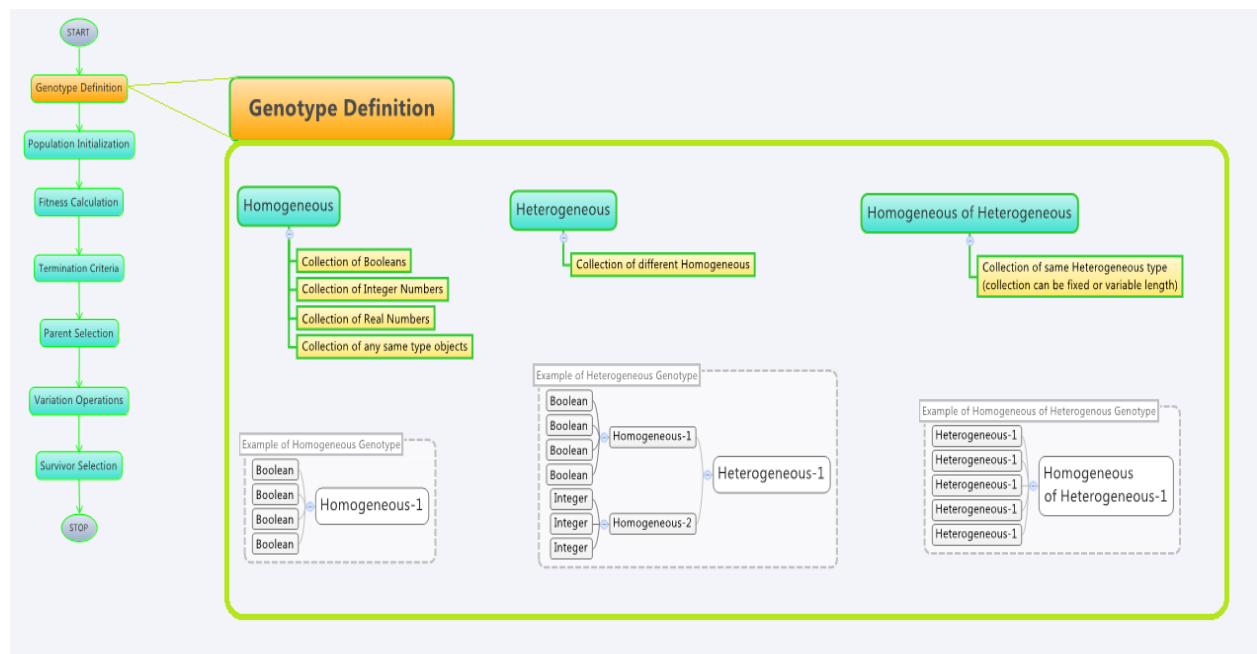


Figure 6. Genotype definition

As exhibited in Figure 6, genotype definition is the first step in the development process: this is shown in the flow chart. The rest of the figure exhibits the relationship between homogeneous and heterogeneous genotypes. The first example (Homogeneous-1) is that of a Homogeneous genotype made of 4 Booleans. The second example (Heterogeneous-1) is that of Heterogeneous genotype consisting of 4 Booleans and 3 Integers. The third example (Homogeneous of Heterogeneous-1) is a homogeneous genotype made of 5 Heterogeneous-1 parts.

The type and structure of genotypes are closely related to variation operations. This will be explained in detail in the variation strategy section.

2.3.2 Population Initialization

Initialization is directly bound to the genotype representation. For every Homogeneous type, CM provides an initialization mechanism at Genotype level.

As exhibited in Figure 7, initialization is the second step in the development process: this is shown in the flow chart. In addition, a random generator engine is used create the initial population. The random generator class is registered during design time either manually or through the CM application and then invoked at runtime by the CM framework.

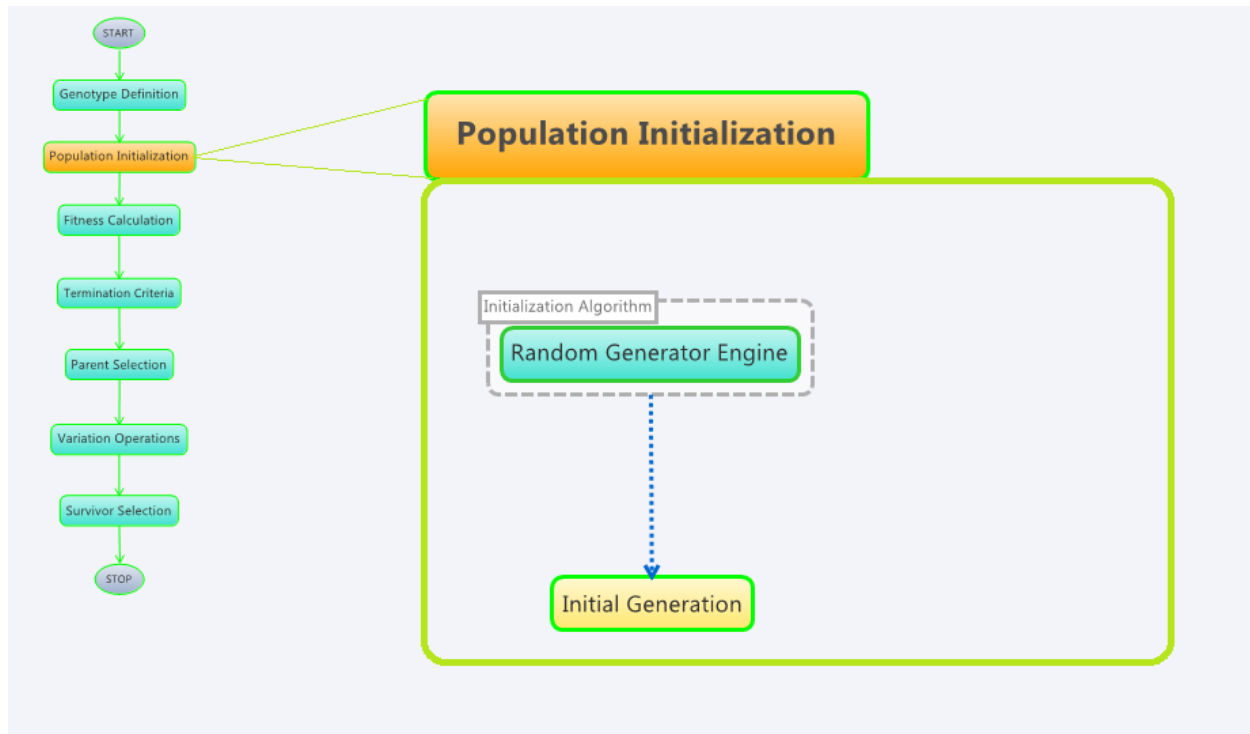


Figure 7. Population initialization

2.3.3 Fitness Calculation

Fitness calculation is dependent on the problem domain. The one thing that all fitness calculations have in common is that they return one or more numerical values. Also, fitness calculation may require communications with sources external to the EA application. To accommodate this possibility, CM allows both internal and external fitness calculation. Internal fitness calculation code must be entered by the user. As to external fitness calculation, CM allows invoking external processes, serializing the population data to those resources, parsing the response from the external resources to retrieve the calculated fitness values and hence, processing (e.g., scaling and combining) the retrieved values- should the user require it.

Figure 8 illustrates that defining fitness calculation is the 3rd step in the development process. It also shows that fitness calculation can be carried out entirely within the CM framework or via a link to an external resource; even if an external resource is used for fitness evaluation, there remains the option of further manipulation of the received fitness values by internal processes.

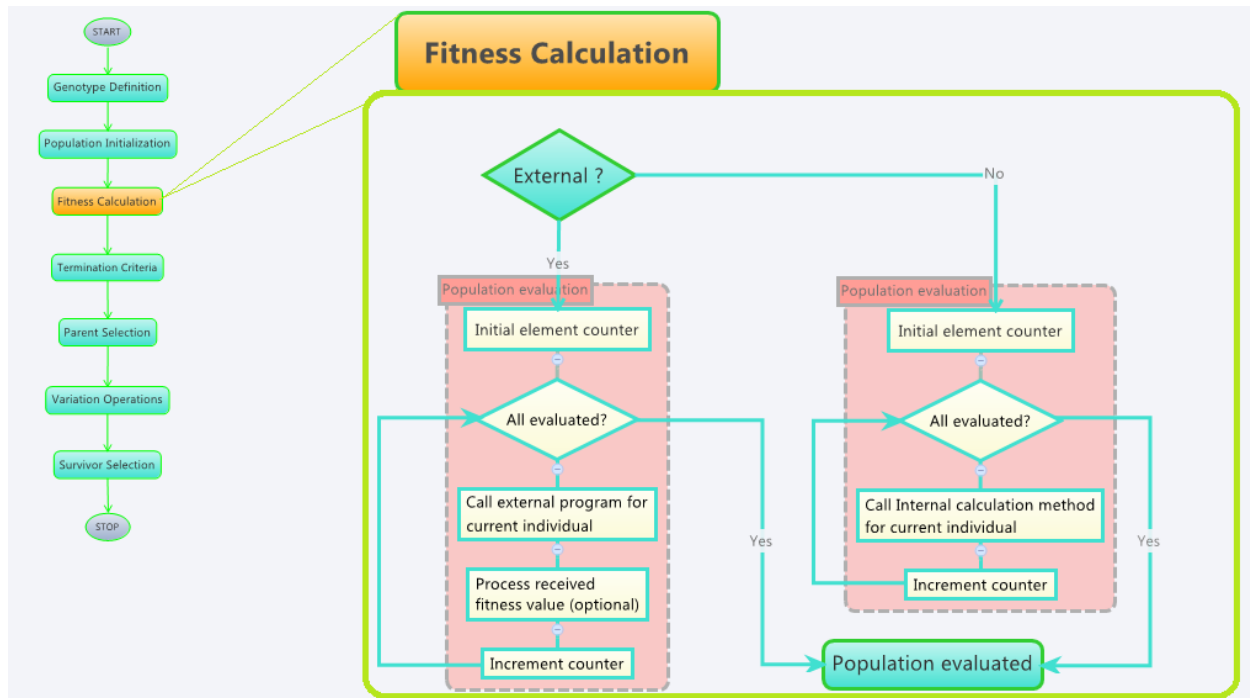


Figure 8. Fitness calculation

If the external option is selected then there are two formats that CM supports for serialization: XML and JSON. The contextual help of the CM application provides concrete examples of how data may be presented in these two formats. Below, we provide arbitrary examples of phenotype serialization in JSON and XML, where the phenotype has a heterogeneous genotype.

JSON Example:

```

{
  "phenome": {
    "genome": [{ "genome": [true, false, false] }, { "genome": [5.913120788
467042, 5.273583533759099, ] } ] },
    "data": null
  }
}

```

Equivalent serialization in XML:

```

<?xml version="1.0"?>
<Phenome>

```

```

    <Genome>
    <Allele>
        <Genome>
        <Allele>true</Allele>
        <Allele>>false</Allele>
        <Allele>>false</Allele>
        </Genome>
    </Allele>
    <Allele>
        <Genome>
        <Allele>5.367599004466042</Allele>
        <Allele>5.821700080361726</Allele>
        </Genome>
    </Allele>
    </Genome>
    <Data>
    </Data>
</Phenome>

```

It is expected for the external process to be able to parse these data, extract the genotype values and return the fitness as in the same format e.g. in case of JSON it could be: {"fitness" : "the value" } and in case of XML it could be : <result name="fitness" value ="the value" /> . There can be multiple entries of name/value pairs in the result. The framework extract all and pass has Java HashMap<String,String> to the evaluate method inside Phenotype for further internal processing.

2.3.4 Termination Criteria

The evolutionary process cannot run infinitely. As shown in Figure 9, CM places all possible termination criteria into three categories, which the user can select from and combine: (a) the Goal Achieved category, which means that an acceptable level of fitness has been attained by at least one individual in the population; (b) the Stagnation Reached category, which means that the improvement in fitness over a preset number of generations is too low to justify continuation; (c) the Resources Exhausted category, which means that a preset limit on a computational resource, such as processing time, has been reached or breached. Finally, the flow chart illustrates the fact that defining the termination criteria is the 4th step in the software development process.

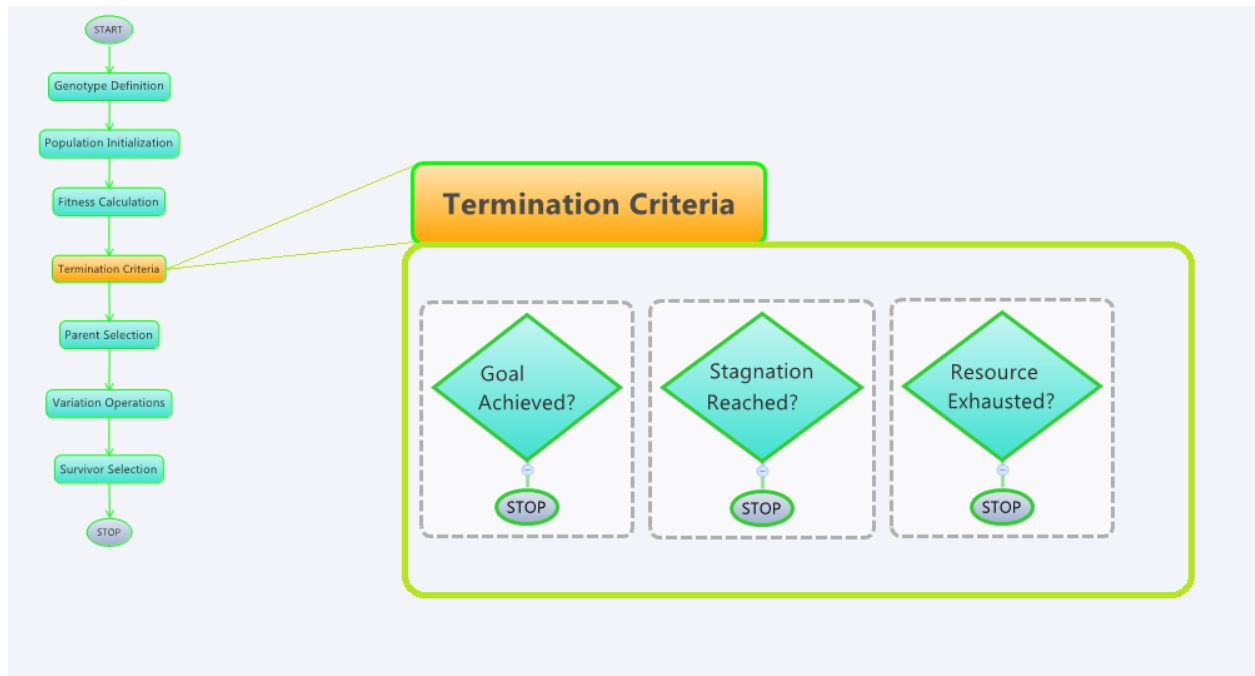


Figure 9. Termination strategy

2.3.5 Parent Selection

Selection shapes the evolutionary process as it influences how the space of possible solutions (individuals) is explored. It is typically independent of an individual's representation, and is a function of an individual's fitness relative to some pool of individuals. Selection occurs at two points in the evolutionary process: when parents are selected from the current generation and when survivors are selected to make the next generation.

During parent selection, one only deals with the current generation, in contrast to survivor selection, where one may deal with both the current generation and the offspring population. Still, the algorithms used for parent selections can also be utilized for survivor selection. We shall cover this relation in more detail but first, we discuss parent selection algorithms.

There is a wide spectrum of algorithms for parent selection [21]. They range from fully deterministic (e.g., Truncation) to fully probabilistic (e.g., Random). They can be applied to two individuals (e.g., Binary Tournament) and to part of, or the totality of, a given population (e.g., Roulette Wheel). Also prior to actual application of any selection method, the 'raw' fitness of an individual can be transformed (e.g., via ranking or scaling) into a different value: it is this new value that is used by the selection method.

Regarding selection methods, a unique contribution of CM is the way it unifies many different parent selection mechanisms into one customizable generic window-based selection mechanism (see figure 10). The selection window can be as large as the whole population or as small as two individuals. Inside the selection window, selection can occur in a deterministic or probabilistic manner. Since fitness is the primary factor in selection, any fitness transformation can also be applied to individuals within the selection window before selection occurs. The output of the window will be deposited into the parent pool. The process of creating windows and extracting parents from it repeats until the parent pool reaches its pre-set size. From this window-based perspective, a Binary Tournament selection can be seen as truncation type selection applied to a window of size two. Also, Roulette Wheel selection can be viewed as probabilistic selection from a window the size of the whole population.

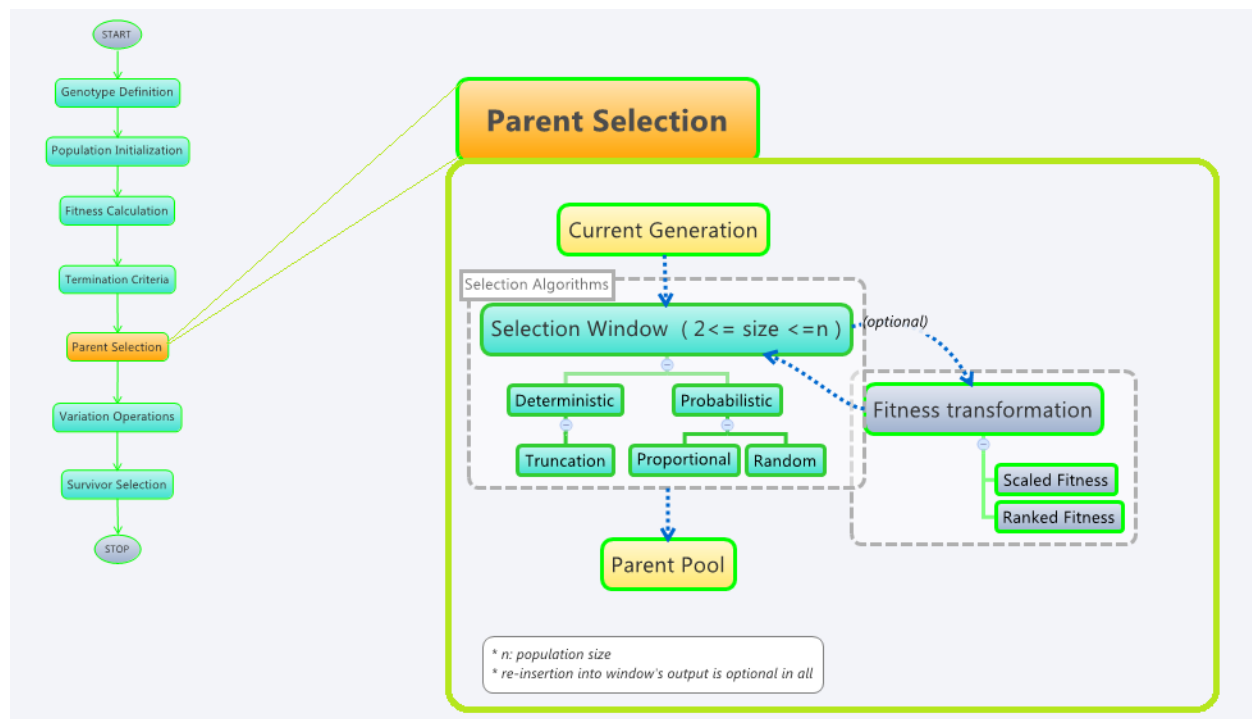


Figure 10. The customizable generic parent selection mechanism adopted by CM

2.3.6 Variation Operations

Variation operations (crossover and mutation) are directly related to genetic representation. In CM's framework the appropriate variation operators are defined for different homogenous genotypes, including Boolean, Integer, Real and Permutation. By definition, crossover occurs at

genotype level while mutation is a gene level phenomenon (except for Permutation mutation, which causes gene shuffling). Hence, the framework provides an assortment of crossover operations for heterogeneous genotypes but no mutation operators.

in case of HofH where genotype consists of heterogeneous blocks, the framework provides crossover operations at top level where heterogeneous blocks can be exchanged. There is no crossover between heterogeneous blocks provided. But inside each heterogeneous block there are homogeneous sub parts where both crossover and mutation operators are available for them. Since HofH also supports flexible number of heterogeneous blocks therefore insertion and delete operators are provided as a type of mutation that affect the size of the HofH genotype.

When it comes to the manner and timing of application of variation operations, different flavors of Evolutionary Algorithms adopt different approaches. For example the canonical GA [22] includes probabilistic crossover followed by mutation. Evolutionary Strategies (ES) is similar but crossover may be skipped [23] and Evolutionary Programming (EP) uses mutation only. Genetic Programming (GP) is very much like GA with tree-style genotypes representing programs. GP has both crossover and mutation but often run them in parallel.

With regard to variation operations (Figure 11), a unique contribution of CM is that any number of variation operations can be used with different probabilities and in different sequencing arrangements along one or more paths linking the parent pool to the offspring pool. This allows users of CM to define a GA-like single sequence of variation operations of, say, crossover followed by mutation, each with its own independent probability. Alternatively, the user can define a GP-style tree of variation operations, with crossover running in parallel to mutation. In this case, the sum of probabilities of all paths must come to 1, and this is checked by CM during development. The definition of the variation operators is the penultimate step in the process of software development, followed only by the definition of survivor selection.

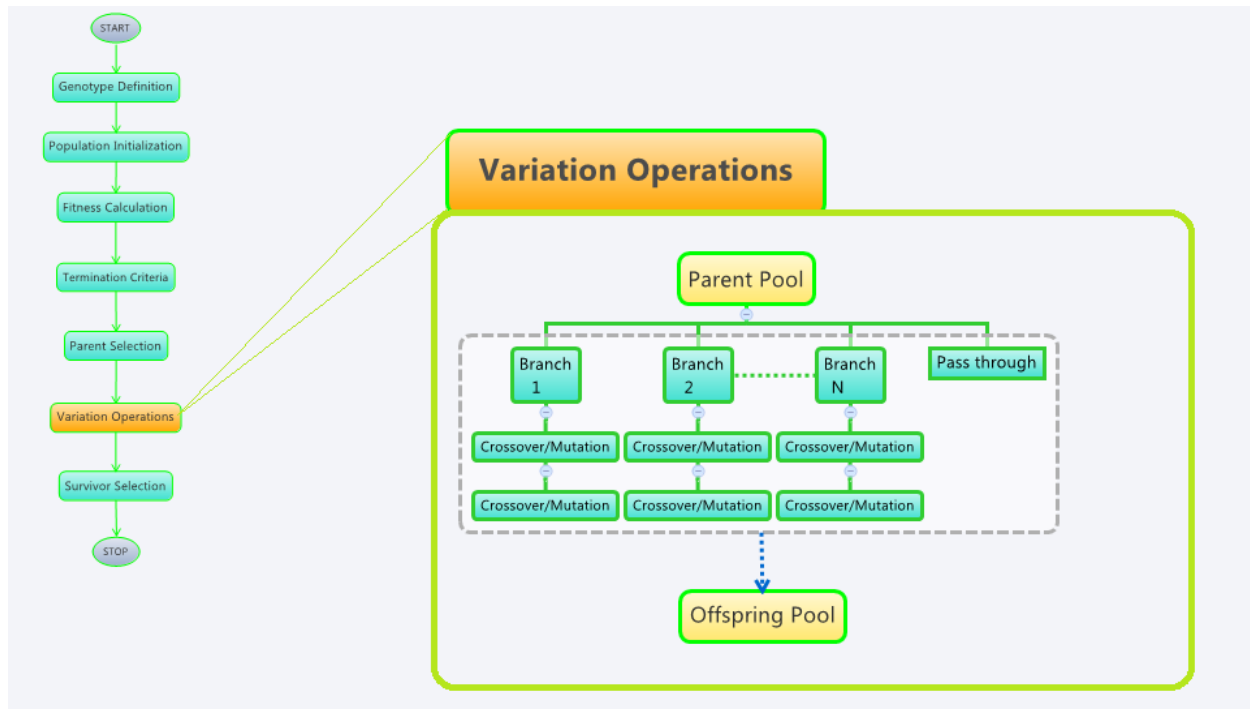


Figure 11. Variation operations

2.3.7 Survivor Selection

Survivor selection creates the next generation using individuals from the current generation and/or offspring pool. With reference to Figure 12, note that all the selection mechanisms that are allowed for parent selection are also available for survival selection. The difference is that we can apply the selection window to any one or both of the current generation pool and the offspring pool or their combined population. In addition, CM allows the use of the special mechanisms of elitism and injection. Elitism is widely used to ensure that the fittest of the current population pass to the next generation. Injection is the re-initialization of part of the next generation, as an added measure of enhancing diversity (and preventing stagnation). Just as it is reasonable to have a low level of elitism, it is analogously advisable to have a low level of injection (if any). In summation, the various means of survivor selection allow up to five different paths from the current population and/or the offspring pool to the next population. This is the final step in the process of software development of an Evolutionary Algorithm using CodeMonkey.

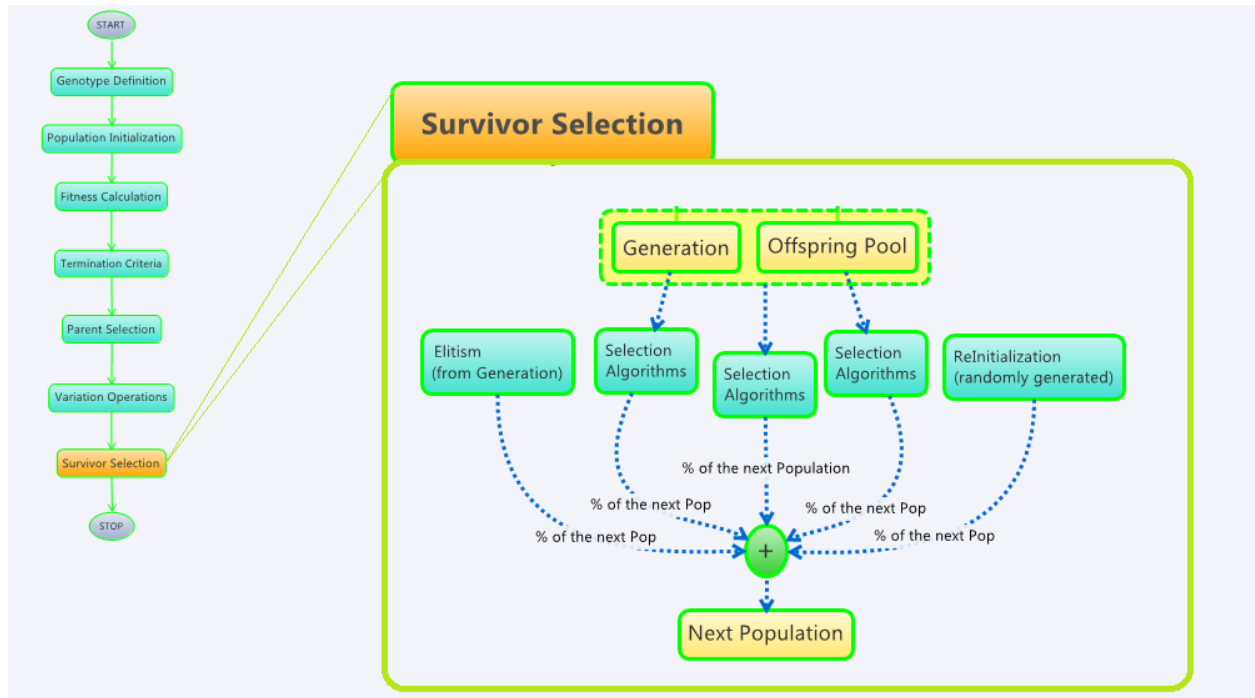


Figure 12. Survivor selection in CM

2.4 Program Execution

Whether the code is generated by the CM application or directly written into the generated program, the *execution* of the resulting Java program follows the process described in figure 13.

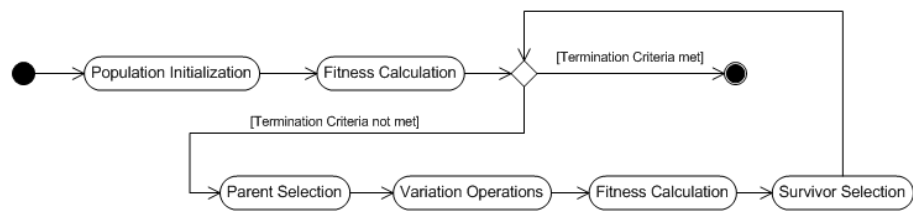


Figure 13. Activity diagram of code execution

First is initialization of the first generation, followed by fitness calculation. Hence, the termination criteria are evaluated. As long as the termination criteria are not satisfied the process goes through parent selection, applying variation operations, evaluating the fitness of the new individuals and generating the next population.

2.5 Use of Eclipse JDT and AST

Eclipse provides powerful APIs (JDT: Java Development Tool and AST: Abstract Syntax Tree) to generate and/or manipulate Java source code and execute it. The CM Eclipse plug-in application makes use of JDT and AST in every step of its interaction with end user. The result is that CM application generates the Java code that runs on top of CM framework for end user. The design and implementation of the framework itself provides help for the plug-in application. Here the integration points between the framework and the plug-in application are described.

- Some of the user interface features of the CM plug-in application are generated on the fly based on available features provided in the framework. For example, the list of available variation operations for each supported data type are computed based on the methods that are defined in the framework. In order to make it as flexible as possible, Java annotations were used inside the framework to explicitly mark the variation methods. Hence, the plug-in implementation, which uses Eclipse AST, is able to parse the framework code and generate the list of available variation operations for each type. There is a significant advantage to this arrangement. If new variation methods are added to the framework (for existing types) then they will automatically be utilized in the plug-in GUI, given that the **@VariationOperator** annotation is added to their declaration. **@VariationOperator** is a custom annotation defined in the CM framework primarily to preserve meta data about variation operators, intended for use inside the CM application. When the CM application builds a list of available variation methods of each type, it dynamically scans the code using JDT, and retrieves a complete list of variation methods, by detecting the **@VariationOperator** at method declarations. Another advantage of this annotation is the 'friendlyName' attribute. It is used in populating the drop-down list for the GUI of the CM application. Since the methods list is generated from the CM framework, if correctly annotated methods are added to the existing types then they will be automatically picked up by the CM application and populated in its GUI.
- The use of registration pattern in the framework allows dependency injection through CM application for registering different strategies. The registration code for these strategies (e.g., parent selection) is added to the generated code based on user input during the various steps of EA software development. This pattern is also useful for expert users

who may want to modify the generated code and register a different implementation for some strategies.

- The framework includes a sub package (`edu.ccil.ec.plugin_template`) that is used as template by CM application. The application creates any new implementation by copying the template code into a user defined package and then modifying it and adding more classes based on user interaction.
- All fitness transformation implementations in the framework are listed in the application GUI- though this is not as dynamic as the listing of variation operators.
- Similarly, all the termination criteria methods that are defined in the framework are present in the application's GUI.

3 Examples of Application

Several experiments conducted to demonstrate the versatility and ease of use of Code Monkey platform. Here we present three of them. The first example deals with mathematical formula, the second and third are examples of combinatorial optimizations. For the first example we provide the detailed step by step interaction with CM application while for the next two we only show the summary of parameters that were given before showing the result.

3.1 The Ackley Function

This function portrays a global optimum surrounded by many local sub-optima. We demonstrate how an end-user can use CM to implement an EA solution to this multi-dimensional multi-modal optimization problem.

3.1.1 Problem Description

The Ackley problem [24] is an n-dimensional minimization problem. The goal is to find the vector $\vec{x} = \{x_1, x_2, \dots, x_n\}$ within $x_i \in \{-32.768, 32.768\}$ that minimizes the function below:

$$F(\vec{x}) = -20 \cdot \exp\left(-0.2 \sqrt{\frac{1}{n} \cdot \sum_{i=1}^n x_i^2}\right) - \exp\left(\frac{1}{n} \sum_{i=1}^n \cos(2\pi \cdot x_i)\right) + 20 + e \quad (1)$$

For this example we use a 10 dimensional space (e.g. n=10).

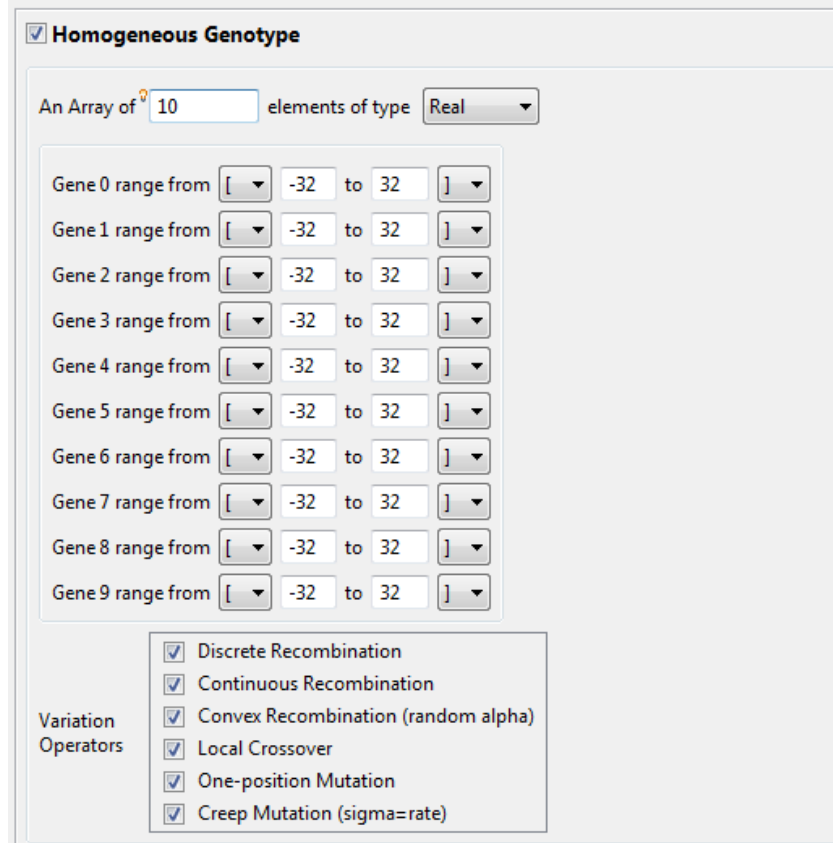
3.1.2 Solution Outline

We start by defining the genotype. In this case it will be a list of real-valued numbers, one per dimension. The dimensions can be initialized randomly to values from a limited range. The fitness function is the formula itself. The termination criteria are a combination of goal achieved, evolutionary stagnation and resource exhaustion. For parent selection and survivor selection many types of deterministic and probabilistic selection methods can be selected. To generate offspring, a number of crossover and mutation operators can be used.

3.1.3 Implementation Details

The process of EA software development via GUI-based configuration is described below.

Genotype Definition. For this problem, we choose a homogeneous genotype of size 10, and set gene type to real. As shown in Figure 14 we select all available variation operators that the framework provides for real numbers.



☒ **Homogeneous Genotype**

An Array of elements of type

Gene 0 range from to]

Gene 1 range from to]

Gene 2 range from to]

Gene 3 range from to]

Gene 4 range from to]

Gene 5 range from to]

Gene 6 range from to]

Gene 7 range from to]

Gene 8 range from to]

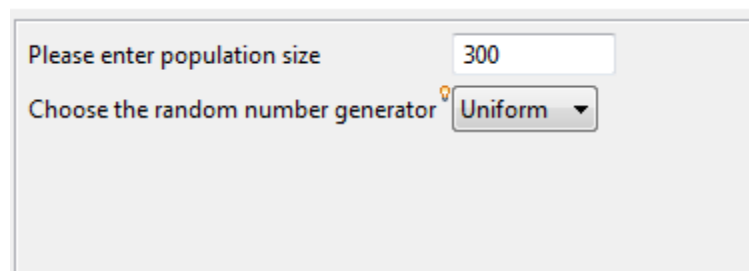
Gene 9 range from to]

Variation Operators

- ☒ Discrete Recombination
- ☒ Continuous Recombination
- ☒ Convex Recombination (random alpha)
- ☒ Local Crossover
- ☒ One-position Mutation
- ☒ Creep Mutation (sigma=rate)

Figure 14. Genotype definition step

Population Initialization. Here, we set the size of the population - which is fixed in CM - and type of initialization. For this application, we set population size to 300, and chose random uniform initialization. Snapshots of the window used to enter the values are shown in Figure 15.



Please enter population size

Choose the random number generator

Figure 15. Population initialization step

Fitness Calculation. Here, we indicate whether the computation of fitness will be done internally or externally, as well as set a few relevant parameters. Optionally, we may enter a value for optimal fitness, and may enter lower and/or upper bounds to the range of fitness values. We also indicate whether the current application is one of minimization or maximization. For this problem, we indicated internal fitness calculation and set the optimal fitness value to 0.0. We provides a Java code fragment reflecting the Ackley function. In fact, CM opens a window that allows the user to input the Java fragment (shown below), then automatically copies it into the right location within the CM framework. Finally, we set the nature of the optimization process to ‘minimization’. Snapshots of the window used to enter the values are shown in Figure 16.

```
@Override
public void evaluate(HashMap<String,String> externalData) {
    Double[] xArr = this.genome.toArray(new Double[genome.size()]);
    this.fitnessValue = external.Ackley.evaluate(xArr);
    return;
}
```

Figure 16. Fitness calculation step

Termination Criteria. Next, the GUI allows the user to select, configure and combine the three conditions of termination. If optimal fitness is known then one can set the Goal Reached condition to a specific absolute or relative value; here an absolute value of 0.0 was entered. If a user wishes to include a Stagnation Reached condition to the termination criteria then he must also enter the lowest acceptable value of relative fitness improvement over a certain number of generations; here we set 3% over 1000 generations. Finally, the user can also include a Resources Exhausted condition in the overall termination criteria, and then set an upper limit to the number of generations or fitness calculations or raw time permissible for evolution; here, we set 3000 generations. These conditions can be combined using the AND or OR logical operators;

for this problem we set an OR combination of all three conditions, as configured above. Snapshots of the window used to enter the values are shown in Figure 17.

☒ **Goal Achieved**

Stopping when within unit of the optimum fitness value of 0.0 unit.

OR

☒ **Stagnation Reached**

Stopping if the best fitness changes by less than % over generations.

OR

☒ **Resource Exhausted**

Stopping when the total reaches

Figure 17. Termination criteria step

Parent Selection. The user is expected to set the size of the selection window (here, 20) and the number of selected individuals (here, 15). He must also chose the type of selection mechanism. We chose fitness proportional selection. Optionally, the user may choose a particular fitness transformation to be applied to (raw) fitness, prior to selection. We opted to transform row fitness into a population rank. The user must also indicate whether replacement (re-insertion of individuals that had already participated in a selection event) is allowed or not. We opted for replacement. Finally, the size of the resulting parent pool must be set (here, 150). Snapshots of the window used to enter the values are shown in Figure 18.

Input

The population size (defined in population initialization step)
300

Selection Process

Enter the selection window input size
20

Enter the selection window return size
15

Choose the type of selection inside the window
Proportional

Selection from the window
With Replacement

Choose the type of Fitness Transformation
Rank

Output

Enter the desired parent pool size
150

Figure 18. Parent selection step

Variation Operations. The next step is defining how the variation operations that were selected during genotype definition are going to be applied to the parents. First, we define the size of the offspring pool and whether replacement is allowed. Hence, we define how many parallel branches are needed and set the probability of each branch. Any residual probability will be automatically assigned to the pass-through path, which does not vary the individual in any way. For each path, one can select any of the available variation operators in sequence and assign each an independent probability. For the current problem, we set the offspring pool to 150 and allowed replacement. Also, we choose two parallel paths, and associated each path with two different crossover and mutation operations, as shown in Figure 19.

The generation size (defined in population initialization step)	<input type="text" value="300"/>		
The parent pool size (defined in parent selection step)	<input type="text" value="150"/>		
Enter the desired offspring pool size	<input type="text" value="150"/>		
Selection from the parent pool	<input type="button" value="With Replacement"/>		
Choose the number of possible variation branches	<input type="button" value="2"/>		
Branch 1			
Enter the probability of going through branch 1	<input type="text" value="0.7"/>		
Choose the Variation method	<input type="button" value="Discrete Recombination"/>	Enter method's probability	<input type="text" value="0.6"/> <input type="button" value="Add More"/>
Choose the Variation method	<input type="button" value="One-position Mutation"/>	Enter method's probability	<input type="text" value="0.6"/> <input type="button" value="Remove This"/>
Branch 2			
Enter the probability of going through branch 2	<input type="text" value="0.3"/>		
Choose the Variation method	<input type="button" value="Convex Recombination (random alpha)"/>	Enter method's probability	<input type="text" value="0.6"/> <input type="button" value="Add More"/>
Choose the Variation method	<input type="button" value="Creep Mutation (sigma=rate)"/>	Enter method's probability	<input type="text" value="0.6"/> <input type="button" value="Remove This"/>
Passthrough Branch			
The probability of passing through without variation (calculated)	<input type="text" value="0.0"/>		

Figure 19. Variation operations step

Survivor Selection. Here, one is required to indicate how much of the next generation comes from (a) the current population, the offspring pool or the two combined. For each of these options, the user must specify the selection method (if any). Also, the user must decide whether and how much of the next population will come through elitism or injection. For our application, we opted for 90% of the next population to come, through fitness proportional selection, from the union of the current population and the offspring pool (without replacement). A further 5% of the next population will come via elitism, with the final 5% coming from injection. Snapshots of the window used to enter the values are shown in Figure 20.

☒ **Current & Offspring Combined** Enter the **Percentage** of output from Current & Offspring Combined **90**

Current & Offspring Combined selection window

Enter the window size **450**

Enter the window's return size **270**

Choose the type of selection inside the window **Proportional**

Choose the fitness transformation type **None**

Selection from the window **Without Replacement**

☐ **Current Generation**

☐ **Offspring Pool**

☒ **Elitism** Enter the **Percentage** of output from Elitism **5**

☒ **ReInitialization** Enter the **Percentage** of output from ReInitialization **5**

Figure 20. Survivors selection step

Open and Run. At this point, CodeMonkey's framework is fully specified and is capable of generating a full Java encoded EA.

The summary of our configuration that was described above through GUI interaction with CM application is presented in Table 2.

Table 2. List of parameters and their values for the Ackley problem

Genotype Definition	Length = 10	Type = Real List
	Lower Bound = -32.0 (for all genes)	Upper Value = 32.0 (for all genes)
	Selected Variation Operator: Discrete Recombination, Continuous Recombination, Convex Recombination, Local Crossover, One-Position Mutation, Creep Mutation	
Population Initialization	Population Size = 300	Random Generator = Uniform

Fitness Calculation	Mechanism : Internal (the formula is manually entered in the code) Optimization Type = Minimization , Target Fitness Value = 0.0		
Termination Criteria	Goal Achieved : Stop once Fitness reached the 0.000001 vicinity of target		
	Stagnation Reached : Stop if progress in Fitness was less than %3 over 3000 generation		
	Resource Exhausted : Stop if number of generations reached 5000		
Parent Selection	Window Input Size = 20 , Window Output Size =15 , Selection Type = Proportional Replacement Allowed , Fitness Transformation =Ranking , Parent Pool Size = 150		
Variation Operations	Offspring Pool Size = 150 , Replacement Allowed , Number of Branches = 2		
	Branch One Probability = 0.7	Operator	Probability
		Discrete Recombination	0.8
		One-position Mutation	0.2
	Branch Two Probability = 0.3	Operator	Probability
		Convex Recombination	0.8
		Creep Mutation	0.2
	Pass through Probability = 0.0		
Survivor Selection	Combined Population (Current and Offspring) = % 90 Window Input Size = 450 , Window Output Size = 270 , Selection Type = Proportional , Replacement Allowed , Fitness Transformation = None		
	Elitism = % 5	Re-initialization = % 5	

3.1.4 Result

From the chart in figure 21 we see that the best fitness reached zero (actual value: 8.88×10^{-16}). This was achieved at the 67th generation. Since the target fitness value was 0.0 and the termination condition had the accepted vicinity of 0.000001 the process stopped at the 67th generation. The genotype of the best individual is a vector of 0s along all 10 dimensions. A time course for the evolution of best fitness and mean fitness is presented in that figure as well.

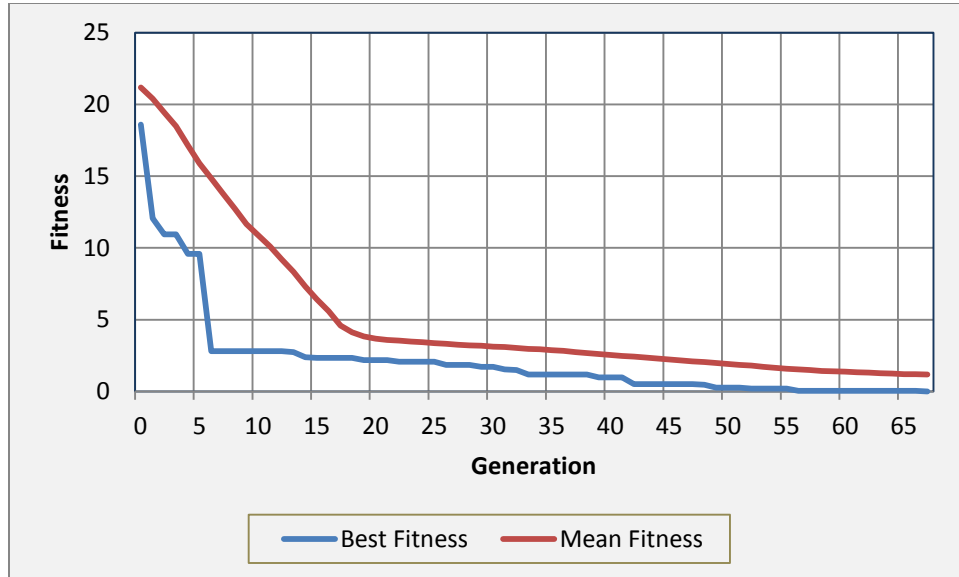


Figure 21. The progress in fitness over generations in Ackley's problem

As it can be seen the mean fitness reduces gradually while the best fitness at first declines sharply as it finds local optima points and then slowly finds the global optima.

The flexibility of the framework allows the user to go back to any of the steps and change the settings for that particular part of the evolutionary process. A re-run of the code will incorporate the changes and affect the results.

3.2 The Traveling Salesman Problem

The Traveling Salesman Problem (TSP) is one of the most studied combinatorial optimization problem in computer science [25].

3.2.1 Problem Definition

For definition let $G = (V, A)$ be a graph where V is a set of n vertices. A is a set of arcs or edges, and let $C = (c_{ij})$ be a distance matrix associated with A . The TSP consists of determining a minimum distance circuit passing through each vertex once and only once. In several applications, C can also be interpreted as cost or travel time matrix.

3.2.2 Implementation

We used the sample data set provided by John Burkardt, with known ideal solutions [26].

Table 3 below summarizes all the information used to configure CM for this application.

Table 3. List of parameters and their values for the TSP

Genotype Definition	Length = 15	Type = Permutation List	
	Selected Variation Operator: PMX Crossover, Order Crossover, Cycle Crossover, Swap Mutation, Insert Mutation, Inversion Mutation		
Population Initialization	Population Size = 300	Random Generator = Uniform	
Fitness Calculation	Mechanism : Internal (the formula is manually entered in the code) Optimization Type = Minimization , Target Fitness Value = not specified		
Termination Criteria	Goal Achieved : N.A. because Target is not known		
	Stagnation Reached : Stop if progress in Fitness was less than %0.1 over 500 generation		
	Resource Exhausted : Stop if number of generations reached 15000		
Parent Selection	Window Input Size = 20 , Window Output Size =15 , Selection Type = Proportional Replacement Not Allowed , Fitness Transformation =Ranking , Parent Pool Size = 150		
Variation Operations	Offspring Pool Size = 150 , Replacement Allowed , Number of Branches = 3		
	Branch One Probability = 0.3	Operator	Probability

		PMX Crossover	0.6
		Swap Mutation	0.2
	Branch Two Probability = 0.3	Operator	Probability
		Order Crossover	0.6
		Creep Mutation	0.2
	Branch Two Probability = 0.3	Operator	Probability
		Cycle Crossover	0.6
		Inversion Mutation	0.2
	Pass through Probability = 0.1		
Survivor Selection	Combined Population (Current and Offspring) = % 90 Window Input Size = 450 , Window Output Size = 270 , Selection Type = Truncation , Replacement Allowed , Fitness Transformation = None		
	Elitism = % 5	Re-initialization = % 5	

3.2.3 Result

The result of running the generated code is shown in the figure 22. The three parameters are Best Fitness (in blue), the mean fitness (in Red) and Sigma (in yellow). Sigma is the standard deviation. It shows how much variation from average exists. Low Sigma indicates data are tend to be very close to mean while high sigma indicates data are spread out across a wide range.

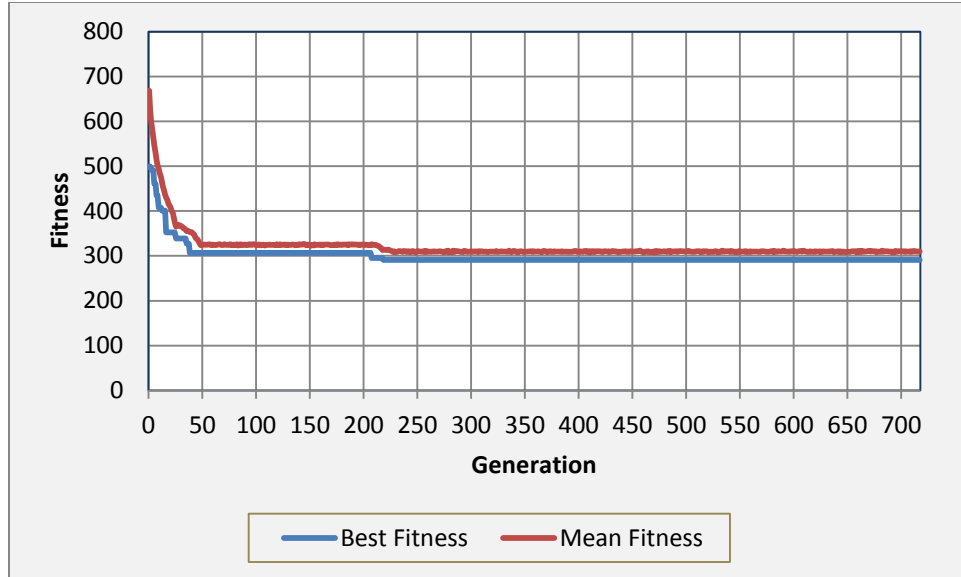


Figure 22. Progress in fitness over generations in TSP

Similar to previous example the mean and best fitness follow the same trend with sharp decline as local optima points are found quickly but the improvement slows down as it tries to find more global optimum. Meanwhile the sigma slightly increases but stabilizes across the process since we have dedicated %5 of population to random injection.

The generated code reaches the optimum fitness that was known for this data set (best fitness = 291.0) at the generation 217. Because no target fitness was given during configuration, the process continues the search until it reaches other termination criteria. In this case with less than %0.1 improvement over the next 500 generation process stops at generation 717. The best overall fitness of 291.0 was obtained by this Genotype= [11, 13, 9, 7, 5, 3, 10, 0, 12, 1, 14, 8, 4, 6, 2]. While this genotype is different than what was defined in the data set the experiment shows that more than one answer exists for this data set.

3.3 The Knapsack Problem

Knapsack is a family of combinatorial problem [27].

3.3.1 Problem Definition

In all variants there are items with profit p_j and weight w_j which are packed into a knapsack of capacity c . The Unbounded Knapsack Problem is the problem of choosing a subset of n items

such that the corresponding profit sum is maximized without having the weight sum to exceed the capacity c . This can be formulated as follows:

$$\begin{aligned} &\text{maximize } \sum_{j=1}^n p_j x_j \\ &\text{subject to } \sum_{j=1}^n w_j x_j \leq c, \quad x_j \geq 0 \text{ integer}, \quad j = 1, \dots, n \end{aligned} \quad (2)$$

Where x_j is an integer variable representing the amount of each item in the knapsack. In the formula, there is no limit on the number of each item but in reality it is limited by the capacity of the knapsack.

3.3.2 Implementation

To have a point of reference and comparison we use the dataset hosted at rosettacode.org [28]. According to the problem description in the rosettacode.org, a traveler leaving Shangri La can pick as many Panacea, Ichor, and Gold as he can fit into his knapsack with the given weight and volume constraints. The exact number is provided in table 4.

Table 4. Unbounded knapsack criteria

Item	Explanation	Value (each)	weight	Volume (each)
panacea (vials of)	Incredible healing properties	3000	0.3	0.025
ichor (ampoules of)	Vampires blood	1800	0.2	0.015
gold (bars)	Shiny shiny	2500	2.0	0.002
Knapsack	For the carrying of	(maximize)	≤ 25	≤ 0.25

Table 5 summarizes all the information used to configure CM for this application. Note that each gene has a different upper boundary based on the weight and volume associated with each item in table 3 above.

Table 5. List of parameters and their values for knapsack problem

Genotype Definition	Length = 3	Type = Integer List (with different range for each gene as shown in Figure. 16)
---------------------	------------	--

	Selected Variation Operator: One Point Crossover, Two Point Crossover, Discrete Crossover, One-Position Mutation and Creep Mutation		
Population Initialization	Population Size = 50	Random Generator = Uniform	
Fitness Calculation	Mechanism : Internal (the formula is manually entered in the code) Optimization Type = Maximization , Target Fitness Value = Not Specified		
Termination Criteria	Goal Achieved : N.A. (because Target is not known)		
	Stagnation Reached : Stop if progress in Fitness was less than %0.1 over 300 generation		
	Resource Exhausted : Stop if number of generations reached 5500		
Parent Selection	Window Input Size = 3 , Window Output Size =1 , Selection Type = Proportional Replacement Not Allowed , Fitness Transformation =None , Parent Pool Size = 40		
Variation Operations	Offspring Pool Size = 30 , Replacement Allowed , Number of Branches = 2		
	Branch One Probability = 0.6	Operator	Probability
		One Point Crossover	0.7
		Two Point Crossover	0.7
	Branch Two Probability = 0.3	Operator	Probability
		Creep Mutation	0.1
		One-Position Mutation	0.1
	Pass through Probability = 0.1		
	Survivor Selection	Combined Population (Current and Offspring) = % 96 Window Input Size = 10 , Window Output Size = 4 , Selection Type = Proportional , Replacement Allowed , Fitness Transformation = None	

	Elitism = % 4	Re-initialization = % 0
--	---------------	-------------------------

3.3.3 Result

The result of running the generated code is shown in the figure 23. This is a maximization problem and fitness climbed quickly to 54500 by 7th generation. As target was not given the process continues and only stopped after additional 300 generations as it was defined stagnation criteria.



Figure 23. Progress in fitness over generations in knapsack problem

This experiment was repeated and it they all the maximum ideal value of 54500. This is in line with the value obtained from deterministic code that is given in the rosettacode.org website. The advantage of using the CM generated code is that it provided different answers with the same optimal fitness. Whereas the deterministic code always returns one answer (0 Panacea, 15 Ichor and 11 Gold). The discovery of more solutions using evolutionary process provides more options as shown in the Table 6.

Table 6. - Unbounded knapsack best found solutions

Solution	# of Panacea	# of Ichor	# of Gold	Total Value	Total weight	Total Volume
Expected	0	15	11	54500	25	0.247
EA alternative	3	10	11	54500	24.9	0.247
EA alternative	6	5	11	54500	24.8	0.247

It can be argued that EA process found better alternatives because they include all three items in the knapsack whereas the deterministic solution only put two items in the bag. Also the weight is slightly less in the alternatives solutions.

3.4 Image Noise Cancellation

To better demonstrate the capabilities of CM, we provide an example of image de-noising using evolutionary computing.

3.4.1 Problem Definition

First, a set of 60 greyscale images from the Berkeley Data Set [29] was compiled. The images are then converted from JPEG to PGM format using ImageJ [30] for easier evaluation. To create a noisy version of these reference images we chose Salt and Pepper noise that is available in ImageJ as shown in figure 24. Salt and Pepper noise is non-Gaussian noise that consists of white and black points randomly scattered over the image. By default, ImageJ adds 5% Salt and Pepper noise, where around half of it is Salt (white) and the other half pepper (black).

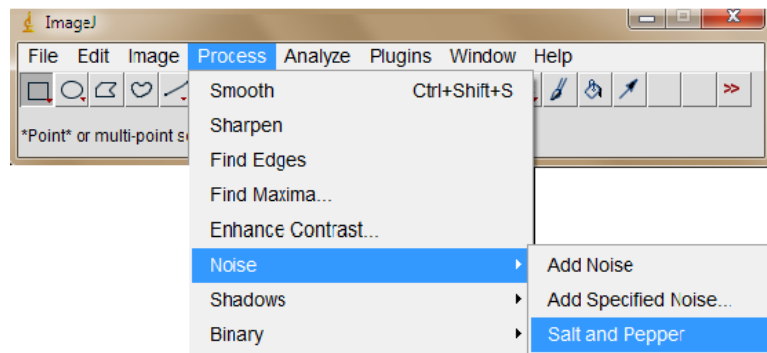


Figure 24. ImageJ salt and pepper noise

For example, figure 25 shows one of the reference images (image #277095) before and after noises is added.

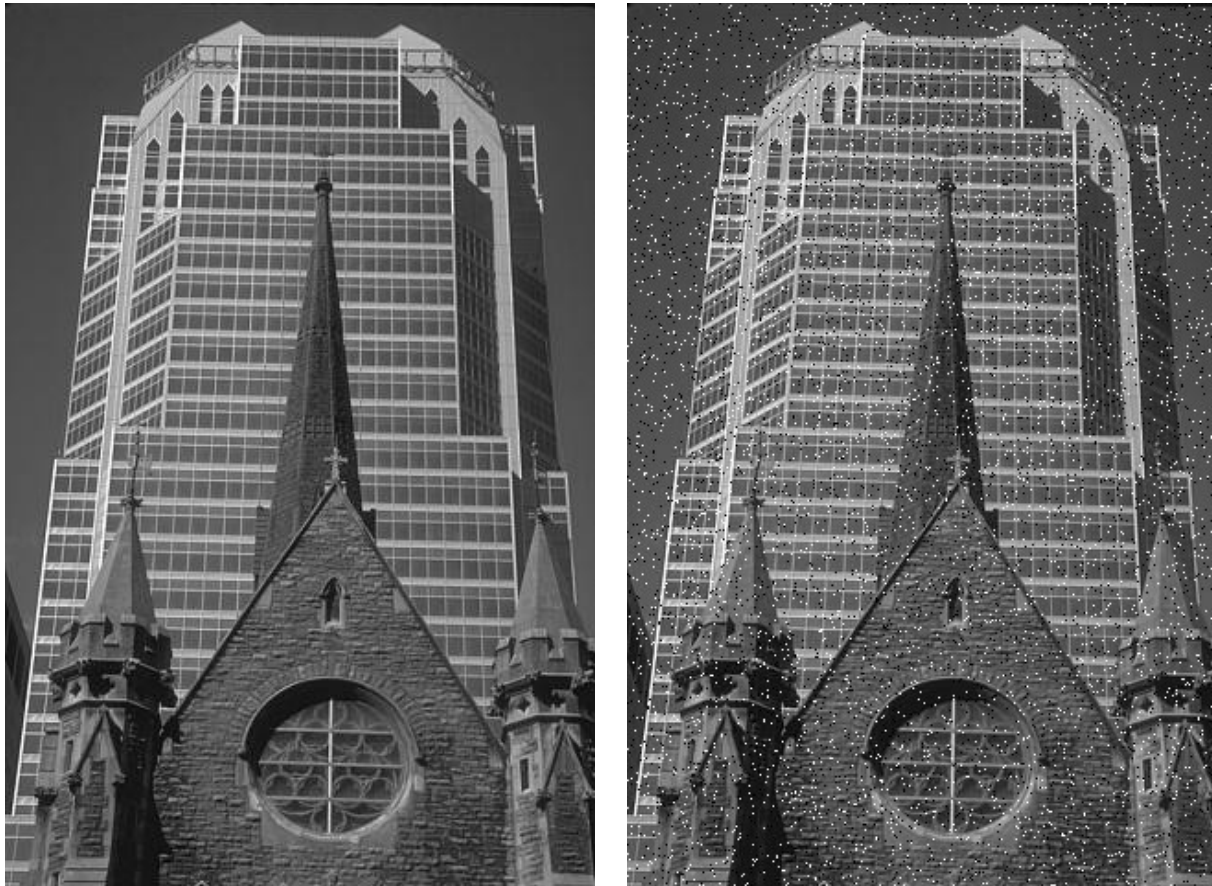


Figure 25. Sample image from Berkley data set, before and after salt & pepper noise

We save the noisy images in PGM format as well to be used as the input to the evolutionary process. The goal is to find a set of Cellular Automata (CA) rules to perform image noise reduction via conditional filtering. In Rosin [31] a sequential floating forward search is used to train the CA rules. Here, we use an evolutionary process to evolve the rule set.

3.4.2 Implementation

For this experiments we are trying to find a set of rules that each acts independently based on the value of a pixel and its neighborhood. We chose to use a 5x5 array around each pixel as it's neighborhood. In this case each rule is a 25 length array of integer values. Each value is in the range of [0,255] which is the color range of pixel in the 8-bit grayscale images.

Each candidate solution contains a set of rules. We chose to limit the length of the set between 2 and 20 rules. The fitness of each candidate is calculated based on randomly selected pixels from first three noisy images that are used as training points. Each training point and its neighbors values are compared to each rule in a given candidate to find possible match. A match is found if the center value of a rule in candidate matches the value of the training point. If no rules satisfies this condition then that training point is not affected by rules in the candidate solution. If however center of more than one rule in the candidate matches the training point we choose the rule that sum of its values is closest to the sum of values of training point and its neighbors (i.e. has least distance). Once a match is found the value of the training point is replaced by the average of values in the matching rule. If no match the training point remains unchanged. To calculate the fitness of a candidate each point is compared with its counterpart in the non noisy version of the image. The difference in their values is calculated. This process is repeated for all training points in all training images and the differences are summed up. The raw fitness of the candidate is this sum after all training points are evaluated. To have penalty for large rule sets the length of the candidate's rule set is added to raw fitness. The following pseudo code shows how up to 1/16 (used as ratio) of points are chosen randomly from a selected image for training.

```

FUNCTION pickCoordinatesRandom(imageHeight, imageWidth, ratio) BEGINS
    //randomly select points from image up to provided ratio
    Coordinates := the empty map    // the map of selected coordinates
    Total := imageHeight * imageWidth
    Needed := Total * ratio
    Points := new empty data set    // to keep the points
    WHILE Size of Points < Needed DO
        P := Random number between 0 and Total
        Add P to Points set
    END WHILE
    FOR every P in Points DO //converting point to x,y coordinate
        X := P / imageWidth    // quotient
        Y := P % imageWidth    //reminder
        Ys := Row in coordinates map of P key
        IF Ys is empty THEN    // new X
            Ys := new empty set
        END IF
        Add Y to Ys
        Put X as key with Ys as value into Coordinates
    END FOR

```

```
    RETURN coordinates
FUNCTION ENDS
```

The goal is to find rule set with minimum fitness value. The following pseudo code demonstrates how the fitness is calculate for a given rule set using training points.

```
FUNCTION evaluate(ruleSet) BEGINS
//for every coordinate in the map get the neighbors & finds the best matched rule, //then apply
the rule if there is match or keep the point unchanged
//calculate error by comparing the output with reference and return sum of errors
errorSum := 0.0
FOR n=0 to n<3 //three is the number of training images
    //For every point in each coordinate map
    FOR Point p:=first training point TO last training point in image n DO
        output := p
        //Get the neighborhood
        neighbors := Read 5x5 Neighbors of p in input image
        //Find the best matching rule (if there is one)
        bestRuleMatchIndex = Find closest to neighbors rule in ruleSet
        //If there is matching rule, apply it to update output
        IF bestRuleMatchIndex > -1 THEN // Match found, apply rule
            output = Average of values in the matching rule
        END IF
        //Calculate the distance of output value of reference value
        error = Absolute difference between output & Reference point
        errorSum := errorSum + error;
    END FOR
END FOR
RETURN errorSum
FUNCTION ENDS
```

The fitness evaluation is performed for all candidates in initial population (300 individuals). The population then goes through proportional parent selection and after that through recombination based on the parameters that were provided to CM. Then fitness of each offspring is calculated with the method described earlier. Finally population goes through survival selection to create the next generation. The process continues until the termination criteria is met. In this case it is

one of the three options. Either a rule set of size 2 with raw fitness of zero is found OR there is no improvement in fitness over 200 generations OR the execution exceeds 7000 generations.

At this point, the best found rule set is applied to all 60 images one by one to generate an output image for each. The Root Mean Square (RMS) error of each output image in comparison to its reference (non noisy) image is calculated. Finally the Average and Standard Deviation of the RMS error values are obtained.

A summary of the parameters and their values for configuration of CM are provided in table 7. In this experiment, we introduced a custom initialization mechanism based on regions of images. Therefore, in addition to writing code for fitness calculation (as required by CM application for the generated Phenotype class), we also added code to the generated Genotype class to fully implement our custom initialization. This shows the flexibility of CM, since users can modify the generated code as needed, hence using a hybrid approach to code generation. We also did a similar customization at the end, to generate PGM images after applying the best evolved rule sets to all the noisy images.

Table 7. Parameters and their values used for the image noise cancelation problem

Genotype Definition	Type = Dynamic List of List Minimum size = 2 Maximum Size = 20		Selected Variation Operators: One Point Crossover, Two Point Crossover, Uniform Crossover, Merge Mutation, Insert Mutation, Delete Mutation
	Genotype Part	Type = Integer List Length = 25 (value range of each gene [0,255]) Selected Variation Operators: One Point Crossover, Two Point Crossover, Discrete Recombination, One-Position Mutation, Creep Mutation	
Population Initialization	Population Size = 300		Random Generator = Uniform
	Population initialized with Randomly selected region of noisy image (code added manually)		

Fitness Calculation	Mechanism : Internal (manually entered formula: sum of errors + genotype size) Optimization Type = Minimization , Target Fitness Value = 2.0		
Termination Criteria	Goal Achieved : Stop once Fitness reached the 0.01 vicinity of target		
	Stagnation Reached : Stop if no progress in Fitness over 200 generation		
	Resource Exhausted : Stop if number of generations reached 7000		
Parent Selection	Window Input Size = 20 , Window Output Size =15 , Selection Type = Proportional Replacement Allowed , Fitness Transformation =Ranking , Parent Pool Size = 150		
Variation Operations	Offspring Pool Size = 150 , Replacement Not Allowed , Number of Branches = 2		
	Branch One Probability = 0.6	Operator	Probability
		One Point Crossover	0.9
		Insert Mutation	0.1
		(Part) Two Point Crossover	0.9
		(Part) One Position Mutation	0.05
	Branch Two Probability = 0.2	Operator	Probability
		Uniform Crossover	0.9
		Delete Mutation	0.1
		(Part) Discrete Recombination	0.9
		(Part) Creep Mutation	0.05
	Pass through Probability = 0.2		
Survivor Selection	Combined Population (Current and Offspring) = % 90 Window Input Size = 45 , Window Output Size = 27 , Selection Type = Truncation , Replacement Not Allowed , Fitness Transformation = None		
	Elitism = % 10	Re-initialization = 0	

To contrast the effectiveness of our evolutionary method of de-noising we apply three different image enhancement filters that are available in ImageJ to the same noisy images. We calculate the RMS errors, their means and standard deviations to compare the result with the corresponding results given by CM.

ImageJ Despeckle Filter. The first ImageJ filter to compare is called despeckle [32]. This filter is a median filter. A median filter is a class of order-statistic filters where filter statistic are derived from ordering (ranking) the elements of a set rather than computing means [33]. In case of the despeckle filter, it replaces each pixel of the image with the median value in its 3x3 neighborhood. The following pseudo-code represents this filter.

```
FOR each pixel in the image
    Extract the 3x3 neighbors around the pixel (for edge pixels there is less)
    Find the median by sorting the extracted values
    Replace the pixel value with the median value
END FOR
```

It is noteworthy that for an edge pixel the immediate neighborhood will be smaller than 3x3. For example, a corner pixel there are only 3 points in the immediate neighborhood. The algorithm takes that into account.

This type of filter is specialized for removing speckles such as salt and pepper noise. Since it is based on medians it is less prone to outliers. The source code of this filter is available at `ij.plugin.filter.RankFilters.java` file, which is part of ImageJ source code at GitHub repository [34].

ImageJ Smooth Filter. The second ImageJ filter is called smooth [35]. It is an arithmetic mean filter [36], which finds the arithmetic average of the pixel values in a $N \times N$ window. It tends to blur the image while mitigating the noise, as it replaces each pixel with the calculated average. In case of ImageJ, its smooth filter is described below:

```
FOR each pixel in the image
    Extract the 3x3 neighbors around the pixel (for edge pixels there is less)
    Calculate the average of the extracted values
    Replace the pixel value with the calculated average
END FOR
```

Same restriction for edge pixels that was mentioned in despeckle filter applies here as well. This filter is also good for reducing salt & pepper type noise but is more prone to the negative effects of outliers. The source code of this filter is available inside `ij.process.ImageProcessor.java` class [37] of ImageJ source code.

ImageJ Remove Outliers Filter. The third ImageJ filter is called remove outliers [38]. It is a conditional median filter. It replaces a pixel with the median value of its surrounding pixels if and only if its value deviates from that median by more than given threshold. So the filter requires three inputs (shown in figure 19). A radius of neighborhood equaling 1 means 3x3 and 2 means 5x5. One also needs to input a threshold value and indicate whether a noise pixel is brighter or darker than the median. For this experiment, we use ImageJ default values as shown in figure 26.

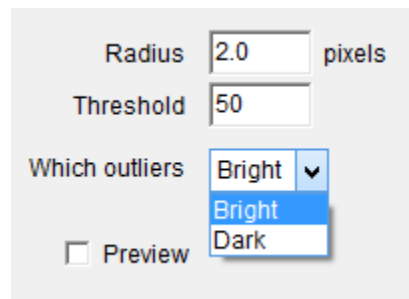


Figure 26. Remove outlier filter input parameters

The pseudo-code for this filter follows:

```
FOR each pixel in the image
    Find the median value within the neighborhood (defined by a radius)
    IF the pixel value + threshold > median value (for dark outliers less)
        Replace the pixel value with the median value
    ELSE
        Do nothing
    END IF
END FOR
```

The code of this algorithm is available in the `ij.plugin.filter.RankFilters.java` class file [34] in ImageJ application source code.

3.4.3 Result

We ran our EA code multiple times and we took the rule set that yielded the lowest mean RMS error as the best found solution. Note that the best rule set does not improve all noisy images equally but that is also the case for ImageJ filters. Figure 27 shows the process of finding the best rule set.

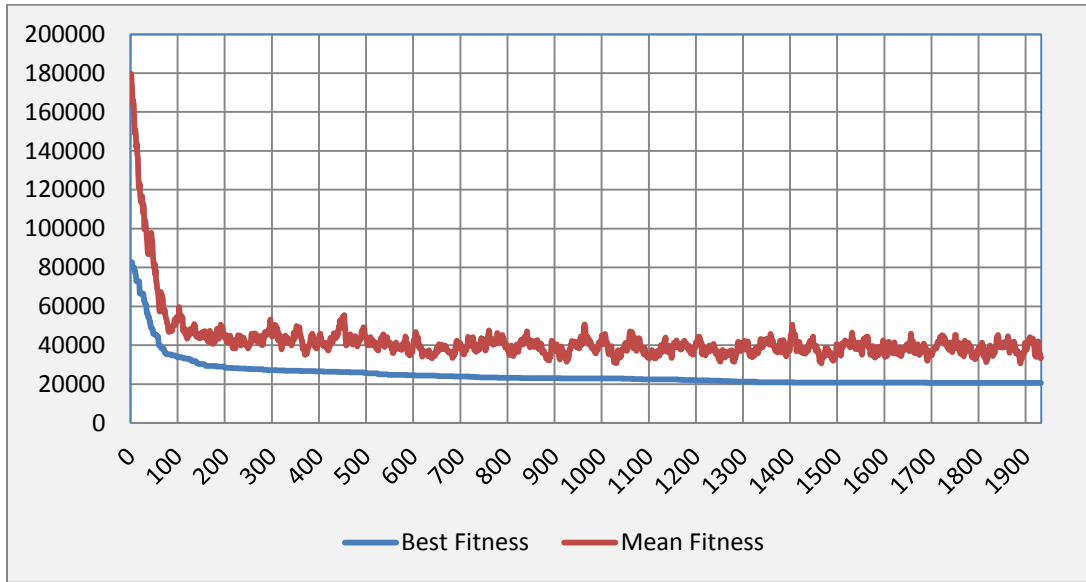
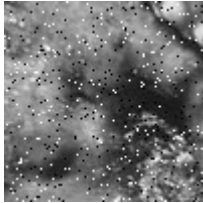
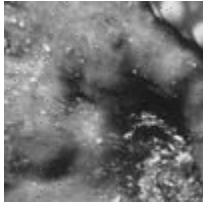
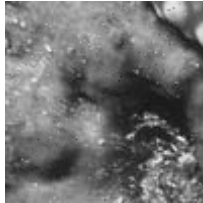
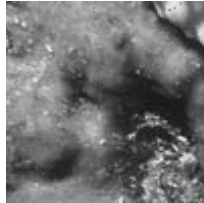
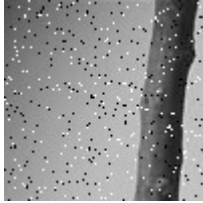
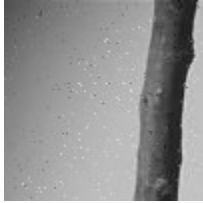






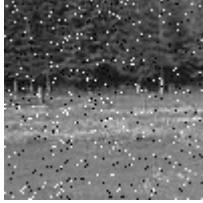





Figure 27. Progress of fitness over the course of the best run of image denoising

The best found rule set contains 9 rules where each is a 25 length array of integer values. After applying this rule set to the noisy images it resulted in average RMS error of 5.828 with the standard deviation of 7.064. In comparison the average RMS error of input noisy images was 31.352 with a standard deviation of 2.573. This means that, on average, there was close to 5 times reduction in noise after applying the evolved rule set. The interesting point is that despite allowing the number of rules in a set to reach 20, the best solution did not contain more than 9 rules. This means that the process did not only look for the best de-noising rule sets but also minimized the number of rules in sets, as it was required by the fitness formula.

Table 8 shows sample output of the best, the median and the least good run of the CM process. While the RMS values varies in multiple runs the visual differences between CM outputs are not significant.

Table 8. Sample RMS and output of best, least and median found solutions by CM

		CM Best Rule Set	CM Median Rule Set	CM least good Rule Set
		Size : 9 rules @ Gen: 1932	Size : 9 rules @ Gen : 1169	Size: 17 rules @ Gen : 1162
Image	S&P Noisy input	Output	Output	Output
12003	RMS= 30.977 	RMS= 4.724 	RMS= 5.941 	RMS= 7.050 
42049	RMS= 31.612 	RMS= 4.364 	RMS= 7.036 	RMS= 10.681 
21077	RMS= 30.466 	RMS= 7.607 	RMS= 10.835 	RMS= 13.430 
28075	RMS= 29.926 	RMS= 4.746 	RMS= 5.588 	RMS= 5.447 

To better demonstrate the results, we also include the output of the run with the least reduction and the one with median output quality. In table 9 we include the average RMS of all three cases and the three ImageJ filters that were applied.

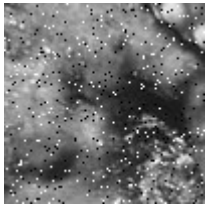
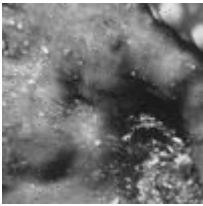
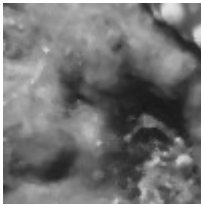
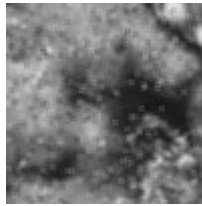
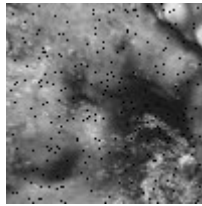
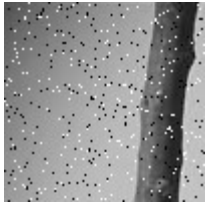
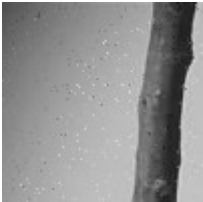


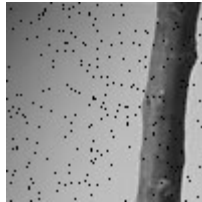
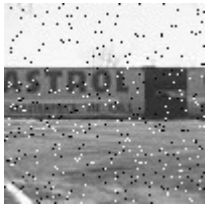




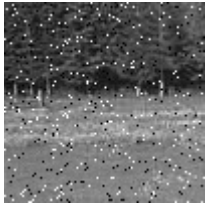



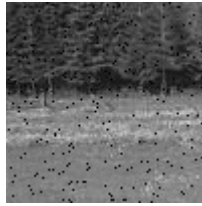
Table 9. Summary of RMS comparison between CM and ImageJ filters

Image Set		Average RMS	Standard Deviation
Input	Salt & Pepper	31.352	2.573
CM Outputs	Best Run	5.828	7.064
	Least good Run	8.834	28.774
	Median Run	7.276	15.624
ImageJ Filter Outputs	Despeckle	9.080	21.275
	Smooth	14.993	11.880
	Remove Outliers	19.241	24.082

The table shows CM has outperformed all three filters of ImageJ. Although this is a comparison of average RMSs, still even the CM output with least improvement was better than the ImageJ filters. To give a more complete picture of the differences in the results, we sampled some of the images from CM output and the outputs of the three filters, and compared them with the each other and the noisy image itself. In table 10 we show a section of these images along with their calculated RMS.

Table 10. Sample comparison of RMS and output images of CM and filters

Image No.	S&P Noisy Input	CM Best Output	ImageJ Despeckle Filter	ImageJ Smooth Filter	Remove Outlier Filter
--------------	-----------------	----------------	----------------------------	-------------------------	--------------------------

12003	RMS= 30.977 	RMS= 4.724 	RMS= 7.848 	RMS= 13.738 	RMS= 19.701 
42049	RMS= 31.612 	RMS= 4.364 	RMS= 5.881 	RMS= 13.662 	RMS= 27.619 
21077	RMS= 30.466 	RMS= 7.607 	RMS= 9.625 	RMS= 15.409 	RMS= 22.282 
28075	RMS= 29.926 	RMS= 4.746 	RMS= 9.817 	RMS= 14.097 	RMS= 17.048 

The comparison shows that applying CM optimized rule sets to noisy images generated lower RMS error relative to the filters. However, ImageJ's Despeckle filter does often return de-noised images that, to the human eye, appear less noisy than CM's filter, though somewhat fuzzy. The main reason is that CM's evolved rule sets are conditional filters that only affect discrete pixels whereas two of ImageJ's filters affect all the pixels, which reduce noise at expense of overall smoothing. Although the Remove Outlier filter is a conditional filter, it did not perform well because the condition is manually pre-set (and fixed). We could have uses this filter multiple times with different threshold values but that would have achieved little beyond accentuating its reliance on manual human rather than automated machine optimization.

4 Summary & Conclusion

CodeMonkey-GA (CM) is a GUI driven software development platform that allows non-experts and experts alike to turn evolutionary algorithm designs into working Java programs, with a minimum amount of manual code entry, usually related to fitness calculation.

Besides its main feature, which is step-wise GUI-driven customization of a generic EA framework, CM has other strengths worthy of note. In terms of representation, CM allows the user to specify a wide variety of homogenous & heterogeneous genotypes. This is done in a manner that allows for automatic identification (by CM) of all acceptable variation operators, to go with the specified genotype. CM has highly configurable definitions for parent as well as survivor selection. CM offers termination criteria that combine any number of three generic & customizable termination conditions. CM has data serialization capabilities, for external fitness computation purposes, which are not available in other platforms. CM is provided as a framework and plug-in application for the Eclipse platform for non-commercial users.

The architecture of CM is based on two parts. One: the framework that contains the core and the implementations of the different genotypes and their associated operators. Two: the application which provides a GUI for end-users to describe their own EA design, terminating with automatic source code generation, in Java.

Through several examples we demonstrated the ease of use and (to some degree) the applicability of the CM application. The Ackley function is a well-known test function for optimization; the Traveling Salesman Problem is a famous example of NP-Complete problems; the Knapsack problem is an example of combinatorial optimization. In all three cases, CM was used to develop working Java programs that provided satisfactory solutions, which were as good as, or better than the given solutions. Critically, in all cases, not a line of code was entered or altered – bar the fitness function – by the user.

5 Future Work

As for all software packages, there is room to extend and improve CodeMonkey. Here, we list some of the areas that we are considering for future releases of CM.

The ability to graphically monitor the progress of evolutionary processing is a useful improvement. Currently, the program logs the key parameters (e.g., fitness) into the console as well as a CSV file, which can be visualized using 3rd party products (e.g., Microsoft Excel). Such charting is, in principle, possible using the Eclipse Chart Engine.

Another area for enhancement is the ability to run the framework on a distributed environment. Since raw fitness calculation can be done in parallel, the ability to compute fitness in parallel would greatly enhance performance and render CM a viable solution for large scale optimization applications.

The scope of use of CM can be extended if support for multi-objective optimization and parameter-less EAs were added to the current version.

Since CM is an open source project we hope it would be embraced by developers who can contribute to the ongoing expansion and improvement of the platform.

References

- [1] M. Ito, F. Zhnng and N. Yoshida, "Collision avoidance control of ship with genetic algorithm," in *IEEE International Conference on Control Applications*, Hawaii, 1999.
- [2] A. Benedetti, M. Farina and M. Gobbi, "Evolutionary multiobjective industrial design: the case of a racing car tire-suspension system," p. volume 10; issue 3, June 2006.
- [3] C. G. Johnson, "Fitness in Evolutionary Art and Music: What Has Been Used and What Could Be Used?," *Lecture Notes in Computer Science*, pp. Volume 7247 pp 129-140 , 2012.
- [4] E.-W. Lameijer, T. Bäck, J. N. Kok and A. P. Ijzerman, "Evolutionary Algorithms in Drug Design," *Natural Computing*, vol. 4, no. 3, pp. pp 177-243, 2005.
- [5] C. S. Krishnamoorthy, P. P. Venkatesh and R. Sudarshan, "Object-Oriented Framework for Genetic Algorithms with Application to Space Truss Optimization," *Journal of Computing in Civil Engineering*, pp. Vol. 16, No. 1, pp.66-75, 2002.
- [6] "Apache Commons Math," 2013. [Online]. Available: <http://commons.apache.org/proper/commons-math/>. [Accessed 03 05 2014].
- [7] "The Genetic and Evolutionary Algorithm Toolbox for Matlab," 2007. [Online]. Available: <http://www.geatbx.com>. [Accessed 03 05 2014].
- [8] "Evolving Objects," 2012. [Online]. Available: <http://eodev.sourceforge.net>. [Accessed 03 05 2014].
- [9] M. Keijzer, J. J. Merelo and G. R. a. M. Schoenau, "Evolving Objects: A General Purpose Evolutionary Computation Library," Le Creusot, France, 2001.
- [10] "PARAllel and DIStributed Evolving Objects," 2005. [Online]. Available: http://www.inesc.pt/~ewgmcd/SW_Talbi.htm. [Accessed 03 05 2014].
- [11] "EASy Specification of Evolutionary Algorithms," 2014. [Online]. Available: http://easea.unistra.fr/index.php/EASEA_platform. [Accessed 03 05 2014].
- [12] "Distributed Resource Evolutionary Algorithms Machine," 2002. [Online]. Available: <http://www.soc.napier.ac.uk/~benp/dream/dream.htm>. [Accessed 03 05 2014].
- [13] "Watchmaker Framework for Evolutionay Computation," 2010. [Online]. Available: <http://watchmaker.uncommons.org>. [Accessed 03 05 2014].

- [14] "Apache Mahout," 2014. [Online]. Available: <https://mahout.apache.org/>. [Accessed 03 05 2014].
- [15] "Gene Expression Programming for Java," 2010. [Online]. Available: <https://code.google.com/p/gep4j/>. [Accessed 03 05 2014].
- [16] "Java Genetic Algorithms Package," 2012. [Online]. Available: <http://jgap.sourceforge.net>. [Accessed 03 05 2014].
- [17] "Java Class Library for Evolutionary Computation," 2013. [Online]. Available: <http://jclec.sourceforge.net/>. [Accessed 03 05 2014].
- [18] "ECJ Project," 2012. [Online]. Available: <http://cs.gmu.edu/~eclab/projects/ecj/>. [Accessed 03 05 2014].
- [19] S. Periyasamy, A. Gray and P. Kille, "The Epigenetic Algorithm," in *IEEE Congrress on Evolutionary Computation*, 2008.
- [20] "Notes on the Eclipse Plug-in Architecture," 2003. [Online]. Available: https://www.eclipse.org/articles/Article-Plug-in-architecture/plugin_architecture.html. [Accessed 03 05 2014].
- [21] D.Dumitrescu, B.Lazzerini, L. Jain and A. Dumitrescu, "Evolutionary Computation," 2000, p. Chapters 3 to 5.
- [22] J. H. Holland, *Adaptation in Natural and Artificial Systems*, University of Michigan Press, 1975.
- [23] Y.-j. Shi, Z.-C. Liu and S. Ma, "An Improved Evolution Strategy for Constrained Circle Packing Problem," in *Advanced Intelligent Computing Theories and Applications*, Changsha, 2010.
- [24] T. Bäck, "Ackley's Function," in *Evolutionary algorithms in theory and practice*, Oxford University Press, 1996, pp. 142-143.
- [25] G. Laporte, "The Traveling Salesman Problem:An overview of exact and approximate algorithms," *European Journal of Operational Research*, pp. 231-247, 1992.
- [26] "TSP P01 dataset," 2012. [Online]. Available: <http://people.sc.fsu.edu/~jburkardt/datasets/tsp/p01.tsp>. [Accessed 03 05 2014].
- [27] S. Martello and P. Toth, in *Knapsack Problems Algorithms and Computer Implementations*, John WILEY & Sons Ltd., 1990, pp. 81-91.
- [28] "Knapsack problem/Unbounded," 2013. [Online]. Available: http://rosettacode.org/wiki/Knapsack_problem/Unbounded. [Accessed 03 05 2014].

- [29] "Berkley Segmentation Data Set," [Online]. Available: <http://www.eecs.berkeley.edu/Research/Projects/CS/vision/grouping/segbench/BSDS300/html/dataset/images/gray/>. [Accessed 08 06 2014].
- [30] "ImageJ," [Online]. Available: <http://rsb.info.nih.gov/ij/>. [Accessed 08 06 2014].
- [31] P. L. Rosin, "Training Cellular Automata for Image Processing," *IEEE Transactions on Image Processing*, vol. 15, pp. 2076-2087, 2006.
- [32] "ImageJ User Guide (Despeckle)," [Online]. Available: <http://rsbweb.nih.gov/ij/docs/guide/146-29.html#sub:Despeckle>. [Accessed 24 07 2014].
- [33] T. Rabie, "Adaptive hybrid mean and median filtering of high-ISO long-exposure sensor noise for digital photography," *Journal of Electronic Imaging*, vol. 13, pp. 264-277, 2004.
- [34] "RankFilters.java," [Online]. Available: <https://github.com/imagej/imagej1/blob/master/ij/plugin/filter/RankFilters.java>. [Accessed 24 07 2014].
- [35] "ImageJ User Guide (Smooth)," [Online]. Available: <http://rsbweb.nih.gov/ij/docs/guide/146-29.html#toc-Subsection-29.1>. [Accessed 24 07 2014].
- [36] S. E. Umbaugh, in *Digital Image Processing and Analysis*, CRC Press, 2011, p. 553.
- [37] "ImageProcessor.java," [Online]. Available: <https://github.com/imagej/imagej1/blob/master/ij/process/ImageProcessor.java#L1359>. [Accessed 27 07 2014].
- [38] "ImageJ User Guide (Remove Outliers)," [Online]. Available: rsbweb.nih.gov/ij/docs/guide/146-29.html#toc-Subsection-29.1outl. [Accessed 27 07 2014].

Appendix A

Code Monkey Installation/Un-installation Guide

Code Monkey consists of a framework and eclipse plug-in application that enables rapid development of evolutionary algorithms using Java language.

Below are the steps to download and install Code Monkey artifacts and their prerequisites.

1. Download Eclipse IDE for Java from Eclipse website <http://www.eclipse.org/downloads/>.
If you already have Eclipse environment skip to step 3.
2. Unpack the downloaded eclipse zip file into a new directory (e.g. Eclipse)
3. Download the Code Monkey plug-in (PGAF2Plug_1.0.0.xxxx.jar) from CM website.
4. Copy the downloaded jar file into the **plugins** subdirectory of your eclipse installation.
5. Start Eclipse application by running eclipse.exe (or .bin in Linux).
6. The Code Monkey icon should be visible in the tool bar as highlighted below

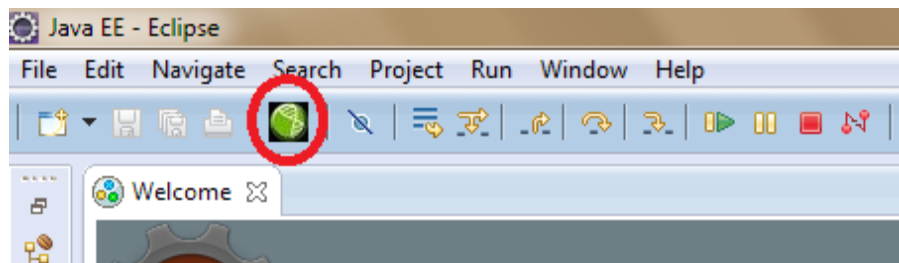


Figure A.28. CM icon in Eclipse toolbar

7. Next you need to import the code base project into Eclipse. First download the project archive file (PGAF2.zip) from CM website.
8. Import the project into your Eclipse workspace through **File->Import** menu
 - 8.1 In the Import popup dialog select General -> "Existing Projects into Workspace" as shown here, and press Next

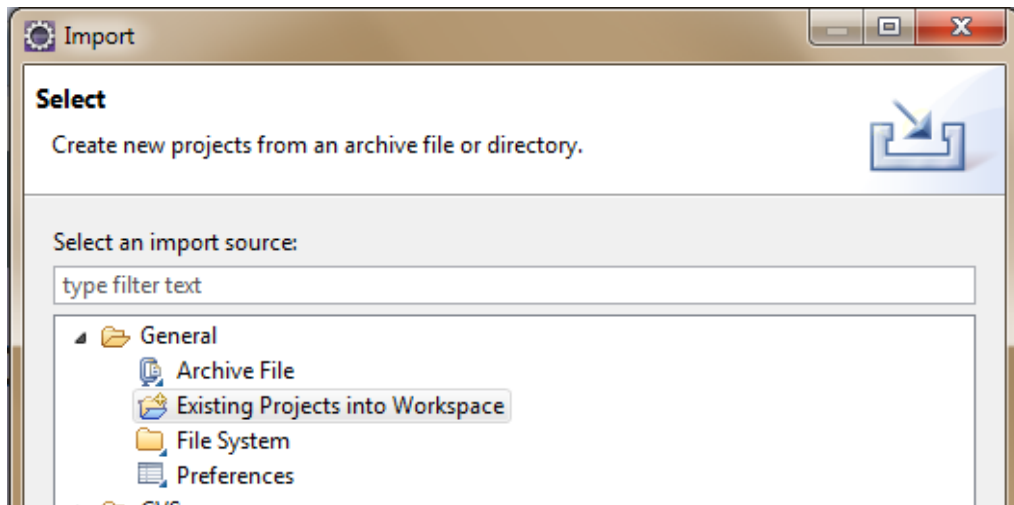


Figure A.29. Import project wizard source selection

8.2 In the new page, select the “Select archive file:” option and browse to the location of downloaded zip file. After project’s name appears under “Projects” section select the checkbox next to the project’s name (PGAF2) and press Finish button.

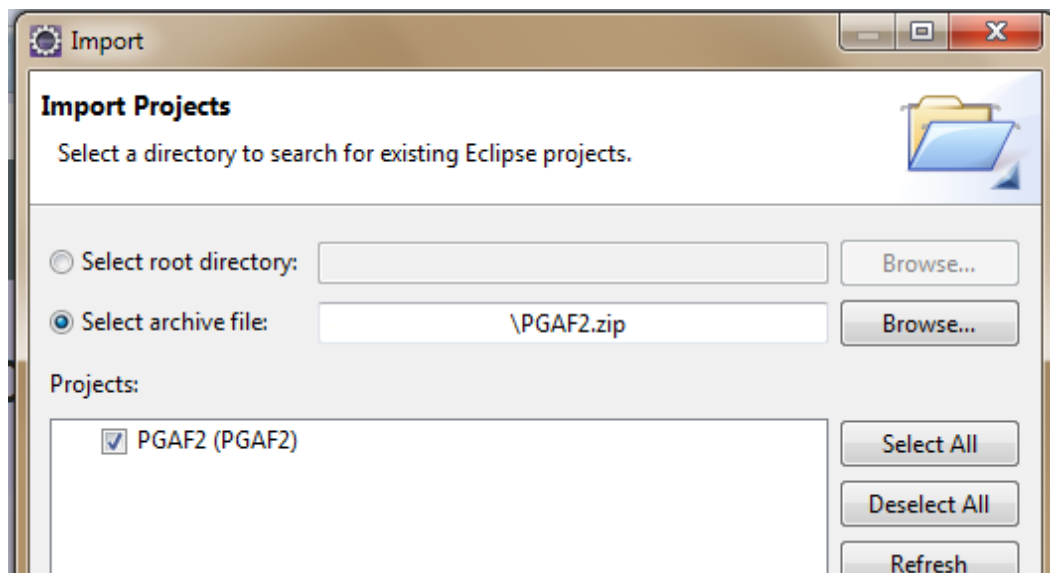


Figure A.30. Import project wizard

9. This completes the installation. The environment is now ready for development of EA application using the Code Monkey plug-in and Framework.
10. Code Monkey application contains a step by step guided navigation on how to generate EA code. To launch the guided navigation (a.k.a. cheat sheet), click on the Code Monkey

icon in the toolbar. Follow the cheat sheets steps to generate the code according to your requirements. For more information read the user guide or watch video tutorials.

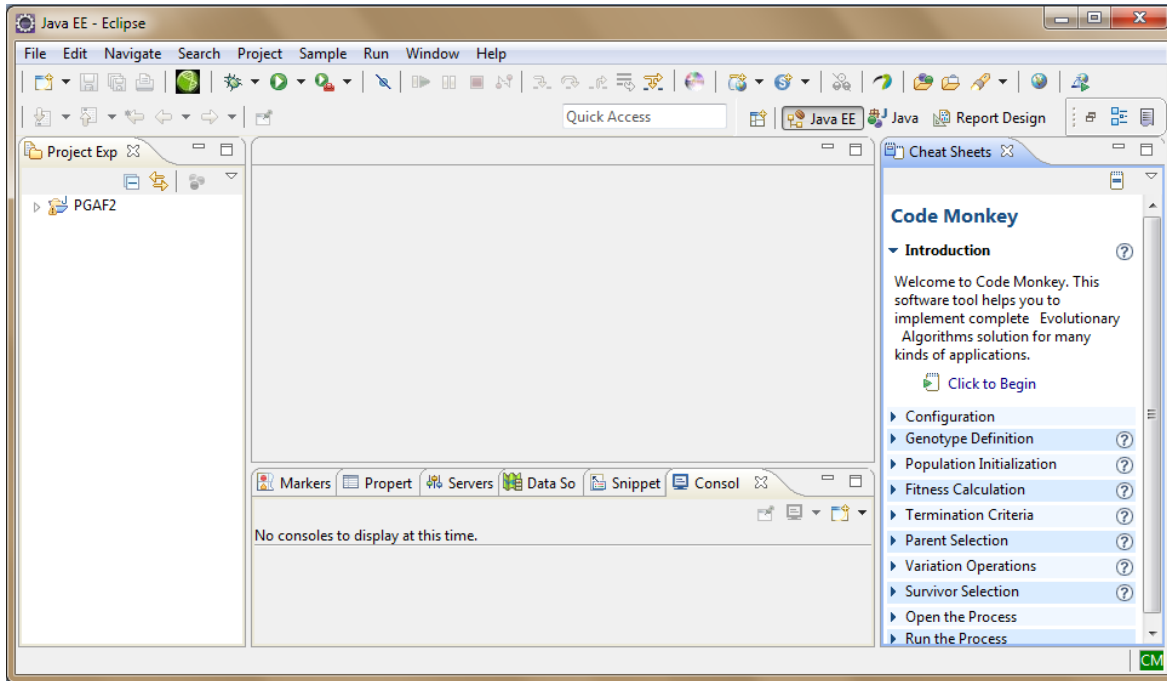


Figure A.31. Eclipse environment after CM installation


To uninstall Code Monkey plug-in application, simply remove the jar (PGAF2Plug_1.0.0.xxxx.jar) from eclipse **plugins** directory and restart the Eclipse. This is also a good practice if you want to install a newer version of Code Monkey plug-in application.

The framework part of Code Monkey is a standard Java project that you imported into Eclipse workspace. It can be removed through Project Explorer in Eclipse.

Appendix B

Code Monkey User Guide

Code Monkey consists of a framework and an eclipse plug-in application that enables rapid development of evolutionary algorithms using Java language. This document guides you through Code Monkey application to generate and execute Evolutionary Algorithm (EA) programs based on your requirements. For installation of Code Monkey please refer to “*Code Monkey Installation Guide*” document.

This user guide is a more detailed companion to the Code Monkey guided navigation help that is available as part of the plug-in application itself. Upon installation of the Code Monkey plug-in application into eclipse you can launch the guided navigation (also known as cheat sheet in eclipse) by clicking on the Code Monkey icon () in the eclipse toolbar. Below is sample page that shows the Code Monkey guided navigation inside Eclipse Cheat Sheet panel on the right.

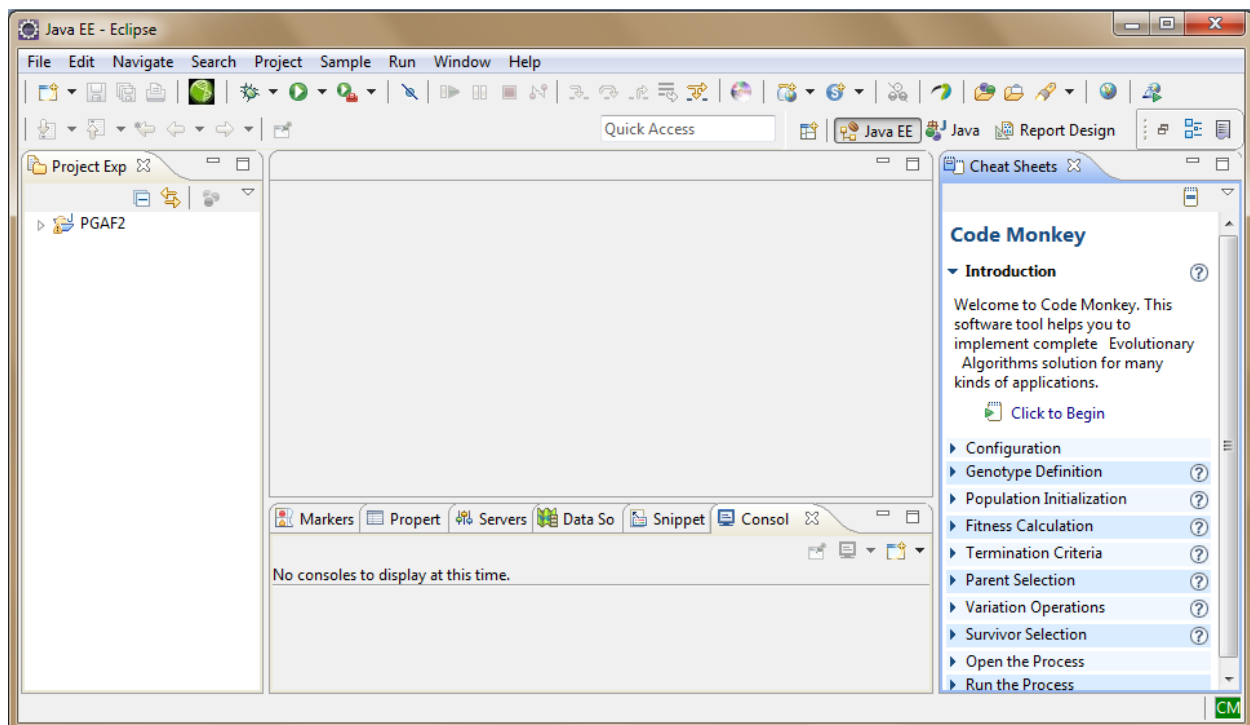


Figure B.32. CM initial page

The guided navigator is self explanatory however in this document we follow the guide and provide more details on each step of the process. In the rest of this document the term 'CM' refers to Code Monkey while 'EA' refers to Evolutionary Algorithms.

1. Introduction

This is the entry point to the process that ultimately generates the EA code. It acts as a welcome message. The actual process starts once you hit the “Click to Begin” at the bottom of welcome message.

2. Configuration

Before any code being generated the process asks you to choose a package name. The package will be created inside the CM framework project and all the will be generated code will reside under this newly created package. The package name can be any valid Java package name and if the package already exists you will be warned to confirm to overwrite. Note that certain package names that conflict with CM framework’s package names will not be accepted.

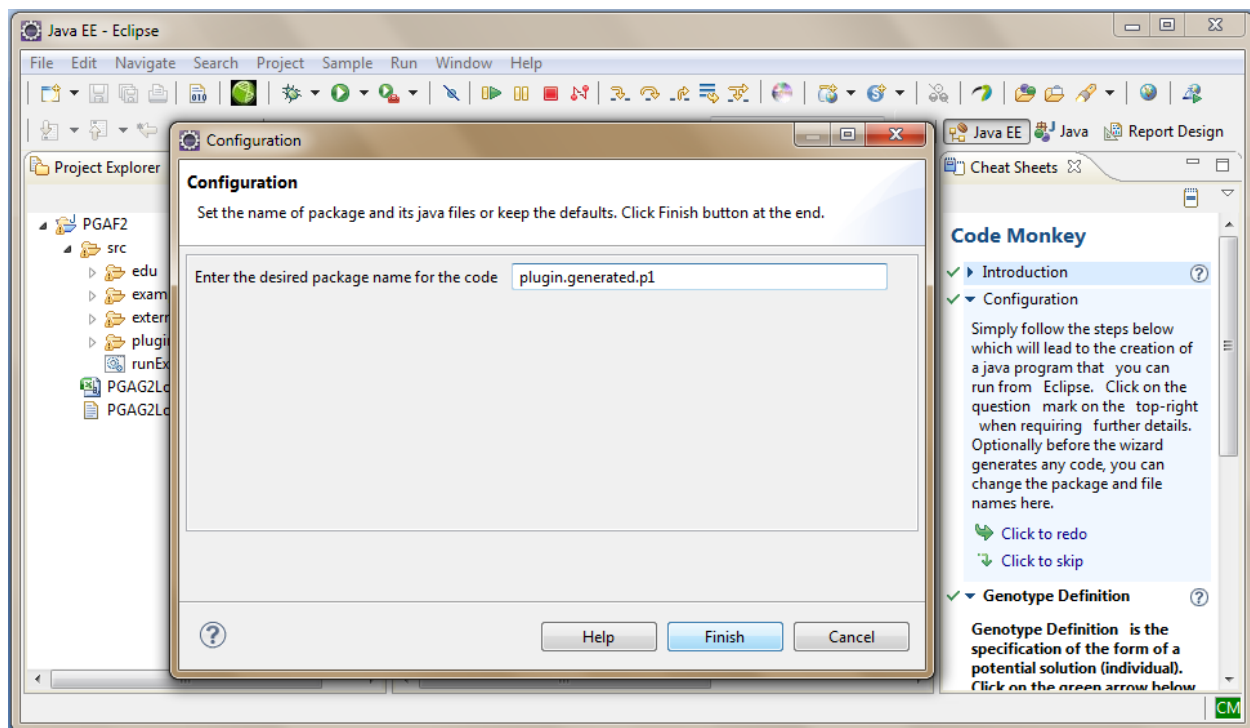


Figure B.33. Configuration wizard

After you entered a valid package name and pressed Finish button the package folder will be created and the next step (Genotype Definition) in the guided navigation panel will expand.

3. Genotype Definition

This step allows you to shape the structure of the genotype that will be created and used inside the package. In total there are three category of genotypes. Homogeneous, Heterogeneous, Homogeneous of Heterogeneous. We describe all three categories in order. The wizard page is preselected with homogenous genotype (where all genes are of the same primitive type) which is also the simplest form of genotype.

If homogenous genotype satisfies your requirements, simply enter the length and then select the type from one of the available primitive types (Boolean, Integer, Real or Permutation).

Note: throughout the wizards you would see a light bulb (💡) or an info (i) icon next to some of the UI input elements. If you move cursor over those icons you will see extra hint about that input or what is expected.

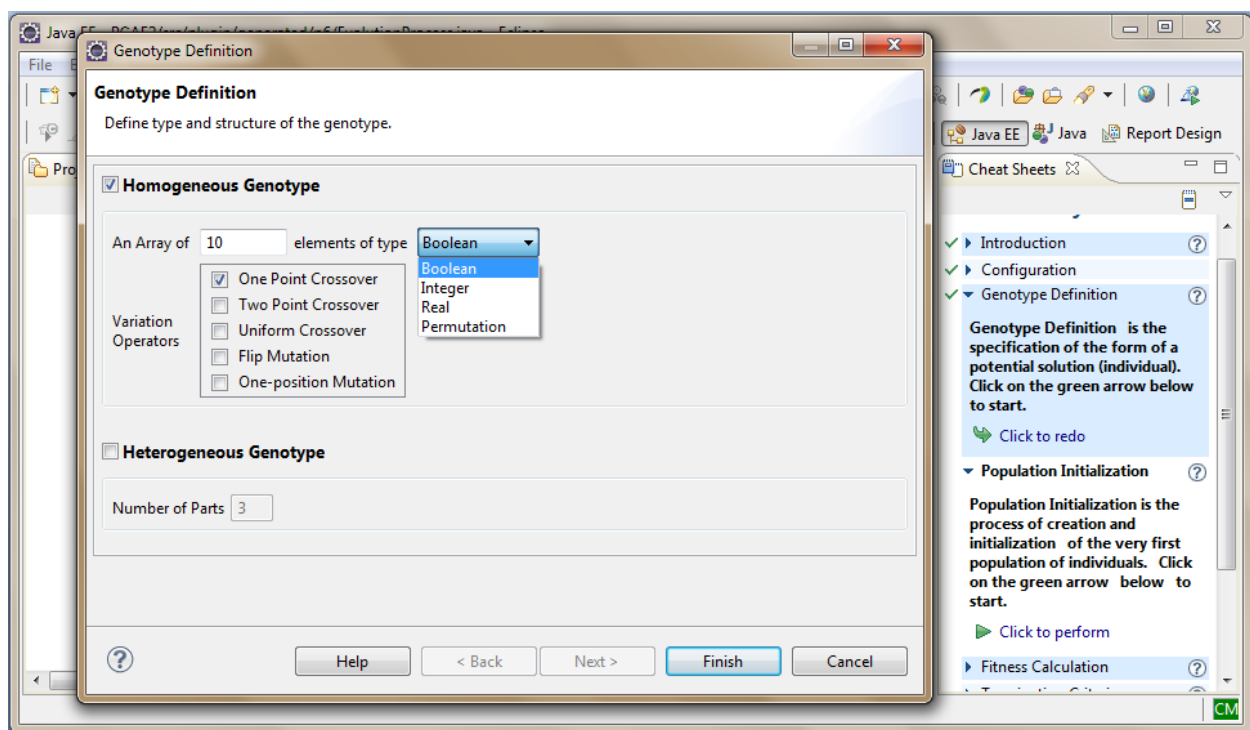


Figure B.34. Genotype definition wizard

If you choose Integer or Real as the type of the genome the wizard expands the section to allow you to enter lower and upper boundary for each gene's value. Since normally the same boundary is used for all

genes the value that you provide for the first gene will be propagated to the rest of empty ranges but you can manually change each boundary separately.

☒ **Homogeneous Genotype**

An Array of elements of type

Gene 0 range from [] to

Gene 1 range from [] to

Gene 2 range from [] to

Gene 3 range from [] to

Gene 4 range from [] to

Figure B.35. Homogeneous genotype

Next you need to select some of the available variation operators that you expect to use for the chosen genotype. If you notice the wizard updates the list of available operators based the primitive type that you select as shown below:

<p>Variation Operators</p> <ul style="list-style-type: none"> <input type="checkbox"/> One Point Crossover <input type="checkbox"/> Two Point Crossover <input type="checkbox"/> Uniform Crossover <input type="checkbox"/> Flip Mutation <input type="checkbox"/> One-position Mutation 	<p>Variation Operators</p> <ul style="list-style-type: none"> <input type="checkbox"/> One Point Crossover <input type="checkbox"/> Two Point Crossover <input type="checkbox"/> Discrete Recombination <input type="checkbox"/> One-position Mutation <input type="checkbox"/> Creep Mutation
Boolean	Integer
<p>Variation Operators</p> <ul style="list-style-type: none"> <input type="checkbox"/> Discrete Recombination <input type="checkbox"/> Continuous Recombination <input type="checkbox"/> Convex Recombination (random alpha) <input type="checkbox"/> Local Crossover <input type="checkbox"/> One-position Mutation <input type="checkbox"/> Creep Mutation (sigma=rate) 	<p>Variation Operators</p> <ul style="list-style-type: none"> <input type="checkbox"/> PMX Crossover <input type="checkbox"/> Order Crossover <input type="checkbox"/> Cycle Crossover <input type="checkbox"/> Swap Mutation <input type="checkbox"/> Insert Mutation <input type="checkbox"/> Inversion Mutation
Real	Permutation

Figure B.36. Variation operators

You have to select at least one variation operator from the list. You will not be asked at this stage to specify the order and probability of these operators. This is the subject of Variation Operations wizard that we will cover later on. However it is recommended to choose as many variation operators as available unless you are certain about not using some of them. You won't be forced to use all selected operators as part of the variation but by selecting them all here you will keep your options open when you reach the Variation Operations wizard.

At this point you have provided all necessary information for your homogeneous genotype. Once you click Finish button the genotype class will be generated.

If you require a heterogeneous genotype (where genes are made of different primitive types) then you must instead select the “Heterogeneous Genotype” checkbox and enter the number of subparts of your heterogeneous genotype based on the number of primitive types that you will need. For example if your genotype consists of a Boolean part and integer part you enter two as the number of parts. Each part will represent a homogeneous genotype itself that you provide their detail in the next page of wizard.

The image shows a user interface for selecting a heterogeneous genotype. It features a checkbox labeled "Heterogeneous Genotype" which is checked. Below this, there is a text input field labeled "Number of Parts" containing the number "2".

Figure B.37. Heterogeneous genotype subpart selection

Once you click Next button you will be directed to the next page of Genotype Definition wizard where you provide the detail of your heterogeneous genotype and its parts.

At the first section you only need to select what variation operators will be used at heterogeneous level (if any). Note that at heterogeneous level only recombination (crossover) operators are available and there is no mutation since mutation is only a gene level operator (except for Permutation type).

For each sub parts of the genotypes the wizard is exactly the same as homogenous genotype wizard that was described earlier. You will choose the size, type and variation operators for each sub part and if applied the gene values boundaries.

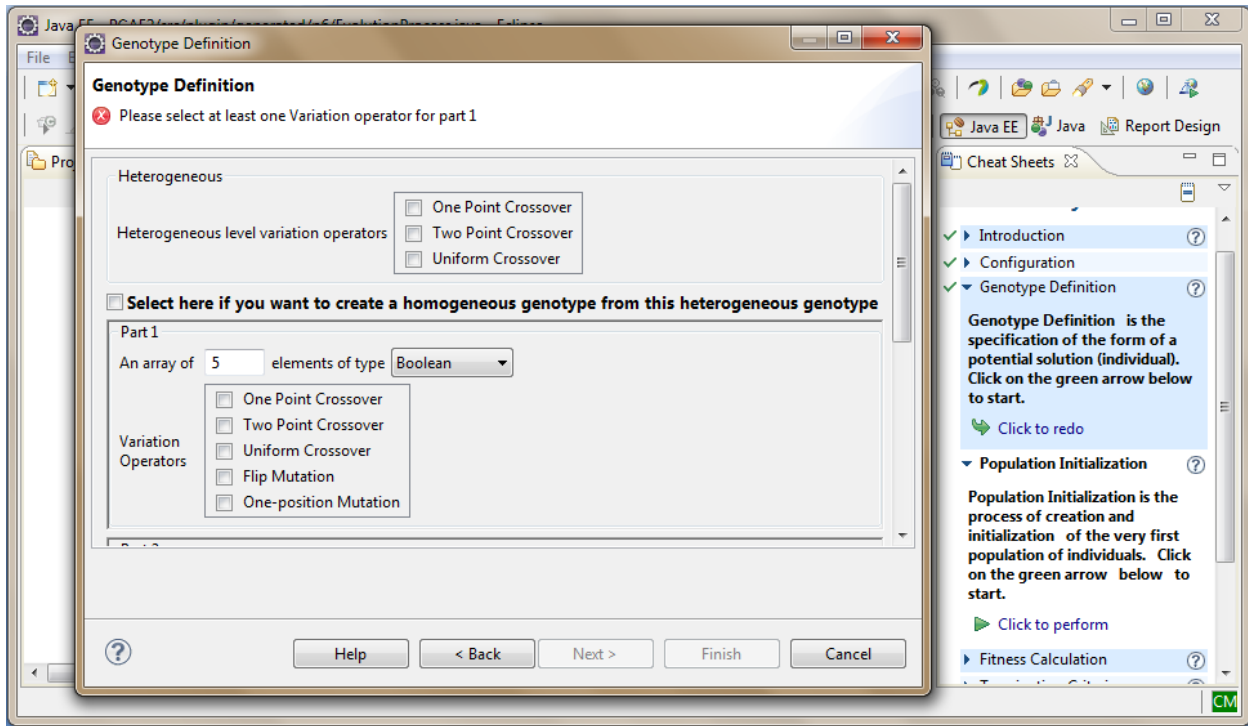


Figure B.38. Heterogeneous genotype wizard

After you entered all necessary information you can press Finish button and the wizard will generate genotype class and its sub parts (one class per each subpart). Note that the heterogeneous variation operator is optional but for each subpart you must choose at least one variation operator. The Finish button will remain grayed out as long as any necessary information is missing. This is a feature throughout the wizard to make sure that each step is completed correctly.

The third type of Genotype which is the most complex one supported by CM application, is a homogenous genotype made of the heterogeneous genotype as its gene. This type of genotype is for the cases where you need a homogenous genotype but a single primitive type is not enough to be used as the gene, rather a combination of them is needed as gene. If that is what desired by your requirements you need to select the checkbox in that indicate this option. The checkbox is below the heterogeneous variation operators section.

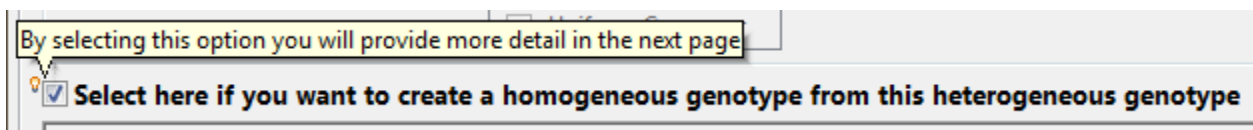


Figure B.39. Selecting homogeneous of heterogeneous option

By selecting this option you will still need to provide information for each subpart of heterogeneous but the whole heterogeneous section will be treated as gene for next step. One difference is that you will not be allowed to specify any recombination operator at heterogeneous level and that section will be grayed out. We provide that option at the next page for the top level genotype. After entering the information of subparts as described earlier you can press Next button to go to the final page.

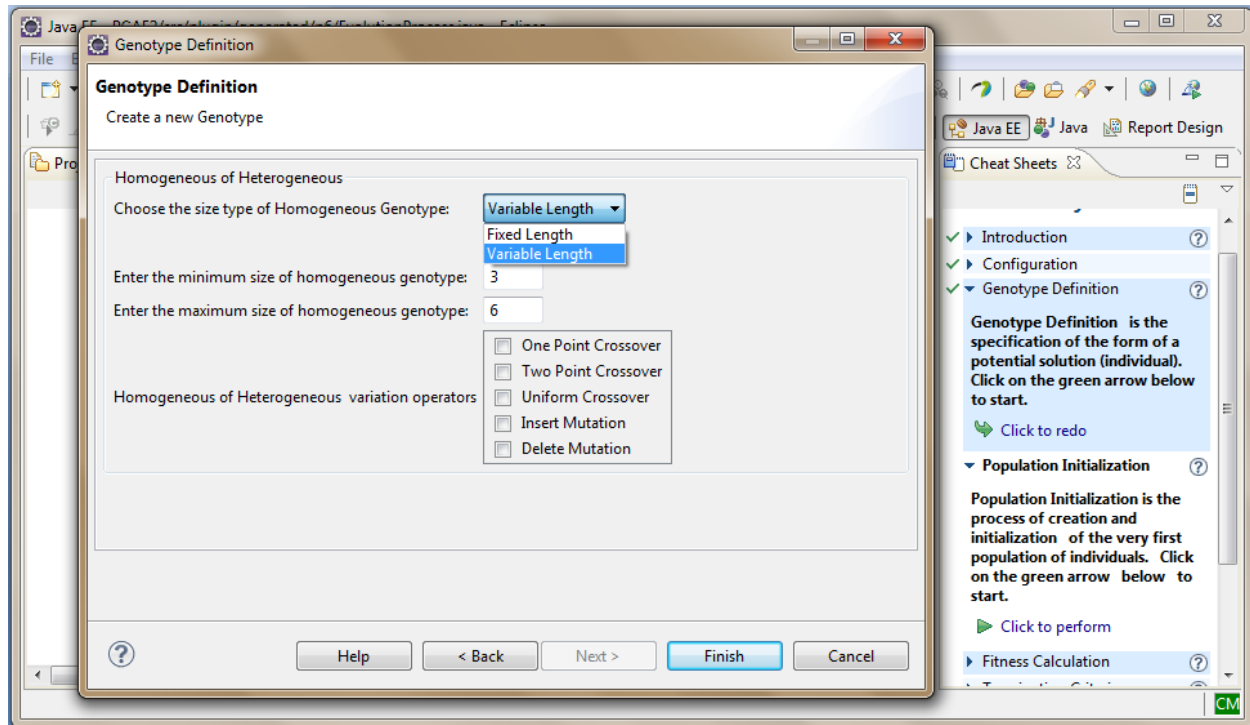


Figure B.40. Homogeneous of heterogeneous genotype definition

At this page you define the structure of the top level homogenous genotype. To provide more flexibility the top level homogeneous genotype can have fixed or variable length depending on your requirements. If fixed length is selected you specify the size and optionally select some variation operators from available list. The variation operators for fixed length genotype are limited to recombination (crossover) since mutation is only possible at gene level (i.e. at each subpart of each heterogeneous unit).

Homogeneous of Heterogeneous

Choose the size type of Homogeneous Genotype: Fixed Length ▼

Enter the fixed size of homogeneous genotype: 5

Homogeneous of Heterogeneous variation operators

- ☐ One Point Crossover
- ☐ Two Point Crossover
- ☐ Uniform Crossover

Figure B.41. Homogeneous of heterogeneous fixed length

If a variable length homogeneous genotype is selected you will need to specify the minimum and maximum size for the genotype instead of fixed length. Optionally you are provided with two extra variation operators (Insertion and Deletion) that allow the size of genotype to change over the course of EA process.

Homogeneous of Heterogeneous

Choose the size type of Homogeneous Genotype: Variable Length ▼

Enter the minimum size of homogeneous genotype: 3

Enter the maximum size of homogeneous genotype: 6

Homogeneous of Heterogeneous variation operators

- ☐ One Point Crossover
- ☐ Two Point Crossover
- ☐ Uniform Crossover
- ☐ Insert Mutation
- ☐ Delete Mutation

Figure B.42. Homogeneous of heterogeneous variable length

Insertion and deletion can be interpreted as mutation at top level since it is a unary operator. In case of insertion a new randomly generated gene (heterogeneous unit) is created and attached to the genotype. And in case of deletion a randomly selected gene (heterogeneous unit) is removed from the genotype. The CM framework will respect the min/max boundaries when applies insertion and deletion.

At this point you are finished with Genotype Definition step. By pressing Finish button a class for top level homogenous genotype will be created along with its heterogeneous unit as inner class. For each subpart of heterogeneous unit a separate class will be generated.

4. Population Initialization

After deciding on the structure of Genotype you need specify how many individuals should be created for the start of the process. This is done in the Population Initialization step where you set the population size. Since all the initial individuals are generated randomly you also choose the type of random generator. Both available options generate numbers with uniform distribution. The difference is in their implementation. The “Secure” option uses newer Java API to generate the random data.

Note that the population size remains constant throughout the process. The size that you provide in this step will be used as reference in following steps to calculate and validate other related numbers. If you change the population size later on you will need to redo steps that use this number as reference. Those steps show the population size as a read only value in their wizard pages.

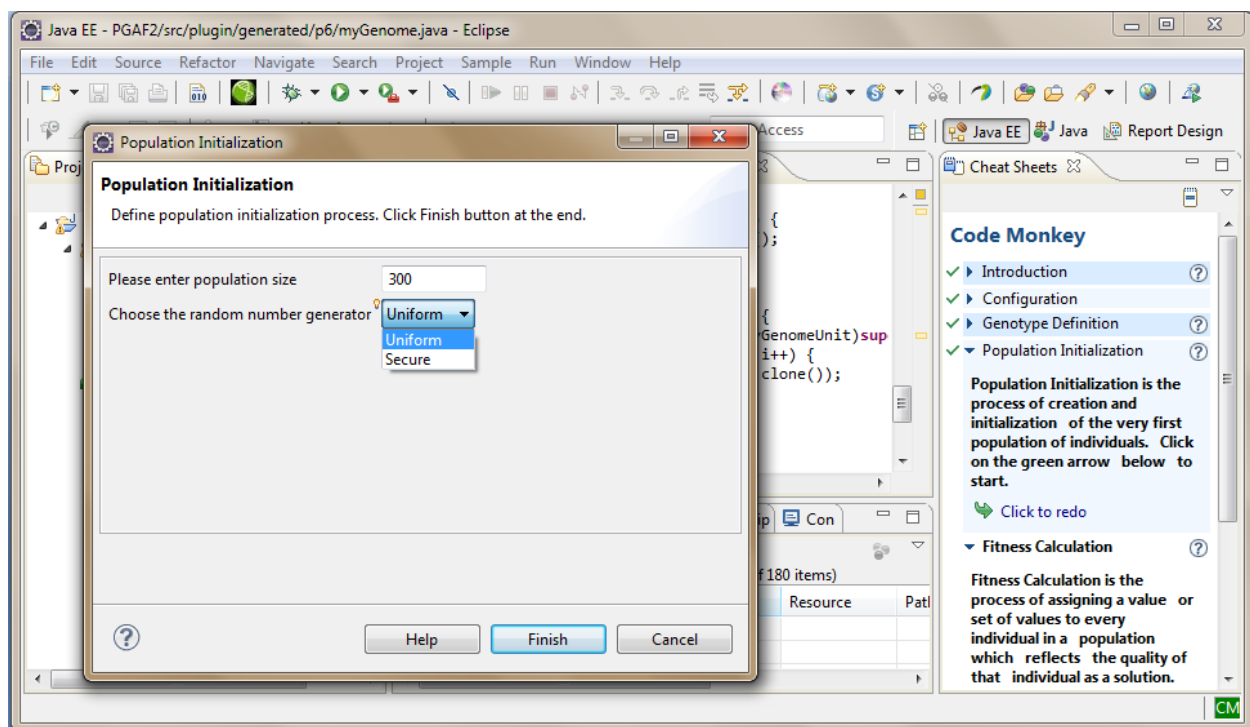


Figure B.43. Population initialization wizard

5. Fitness Calculation

In this step you specify how the fitness of each individual will be calculated. There are two options to choose from when deciding on fitness calculation: Internal or External. The main difference is that with external option you execute an external program to help with fitness calculation. Although this requires serialization of data between external program and CM that we explain further.

Regardless of Internal or External calculation, you can optionally set the target value (the optimal fitness) or lower/upper boundary of fitness, if any of these values is known in advance. These values will be useful when you are determining the termination conditions. For example if you want the EA process to stop when it reaches the exact target fitness or its vicinity.

More importantly you need to define whether this is a minimization or maximization process. This sets the direction of selection process and is specifically needed when the target fitness is not known.

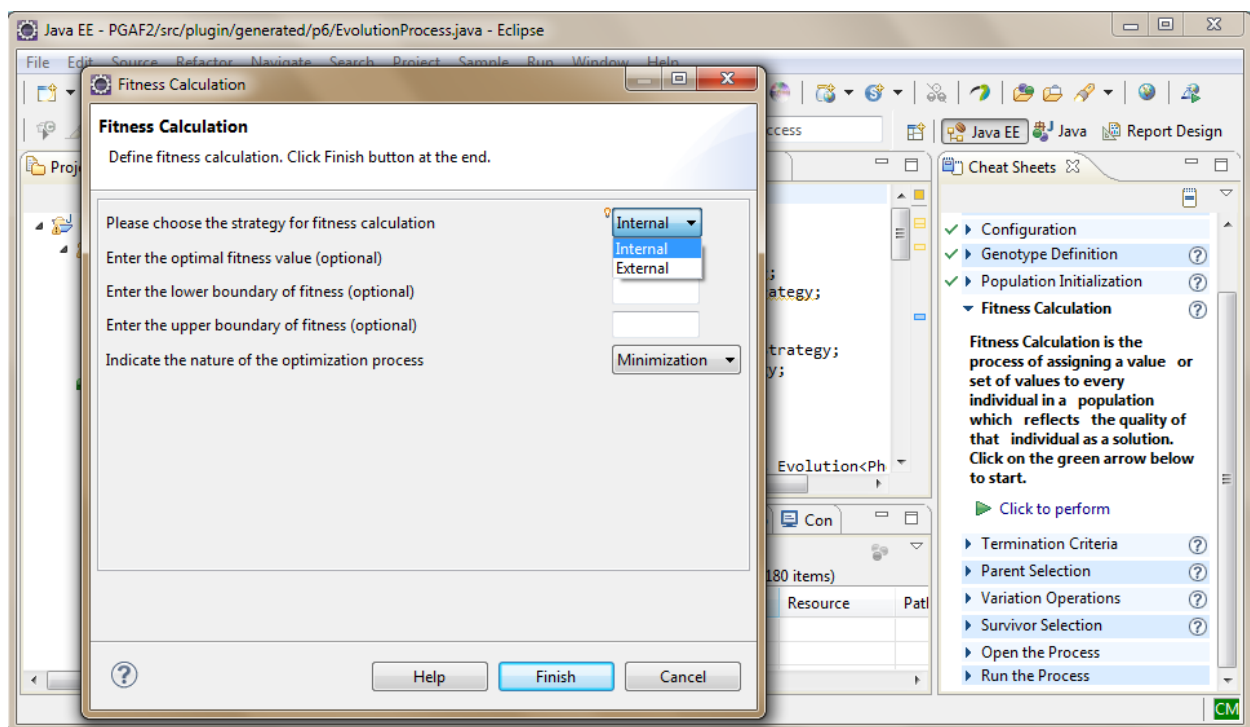


Figure B.44. Fitness calculation wizard

If you choose External strategy the wizard will ask you to provide three more parameters. First is the path-filename of the external program that is needed to run. Optionally provide the folder that program should use as its working directory if it different from the program's location and finally the format to be used for serialization of data between CM framework and external program.

Please choose the strategy for fitness calculation: External

Enter the optimal fitness value (optional): 0

Enter the lower boundary of fitness (optional):

Enter the upper boundary of fitness (optional):

Indicate the nature of the optimization process: Minimization

Enter the external program name (full path):

Browse...

Enter the working directory for the above external program (optional):

Browse...

Choose the format which phenotype objects will be sent in: XML

XML

JSON

Figure B.45. Fitness calculation wizard detail

The assumption is that the external program already understands the CM data format and is able to parse the incoming data and send the result in the expected format. Examples of the data format are available in the guided navigator help.

Once you click Finish button the wizard opens a source file (the Phenotype) in the editor. Note that this is the only place where you are required to manually code. The reason is that there is no way to generalize the logic of fitness calculation. The logic for fitness calculation is required inside the “*evaluate*” method. If you do not provide any code the EA process will not be able to calculate and compare the fitness value of individual candidates and will fail to run.

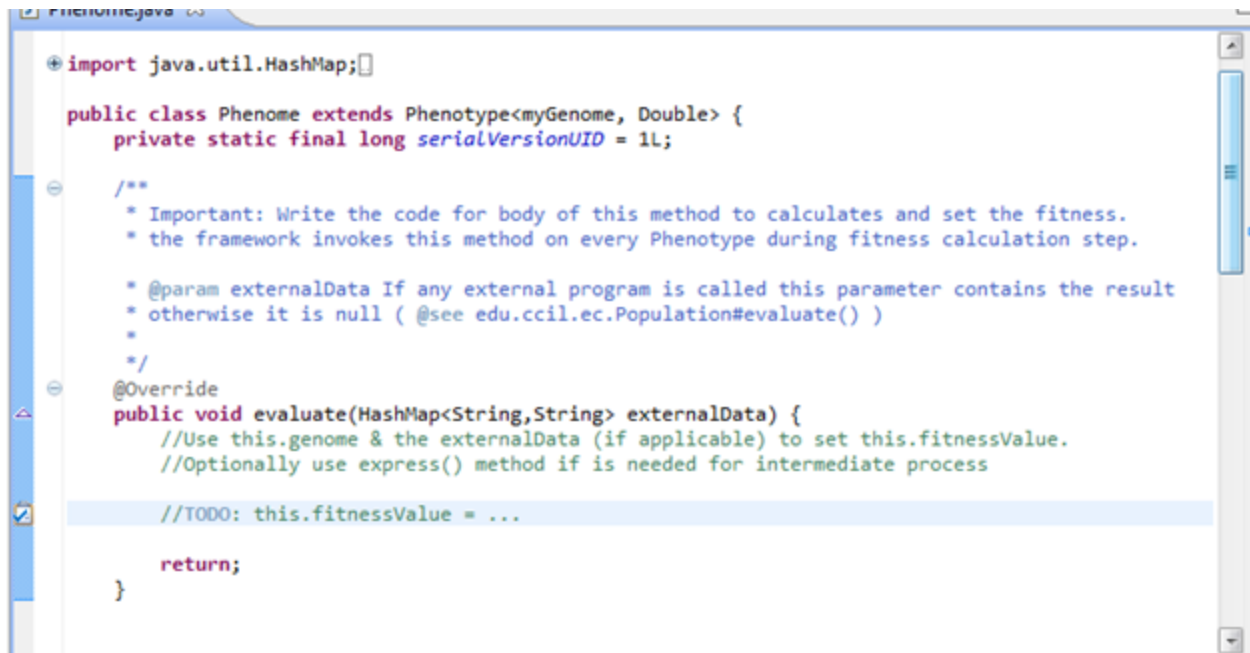


Figure B.46. Phenotype source file for fitness code entry

If you have selected the External option then the data from the execution of external program is available to you as the input parameter “externalData” of the “evaluate” method. Using the external data and the genome itself you should be able to calculate and set the fitness value of phenotype. Note that fitness value is expected to be of type Double (Real number).

After entering the fitness calculation logic save the file and proceed with the next step in guided navigator.

6. Termination Criteria

Any EA process needs termination conditions. In this step you choose a combination of up to three categories. “Goal Achieved”, “Stagnation Reached” and/or “Resource Exhausted”. We will describe them in the same order.

If you have provided target fitness or fitness boundaries you can select the “Goal Achieved” option and define the acceptable vicinity. For that, enter the acceptable delta from Absolute target value that you specified earlier. Or if you have provided the lower/upper boundary you can choose “Normalized” option instead of “Absolute” and enter that vicinity as percentage. During the EA process once any individual reaches that vicinity the process will stop.

Since the EA process may never reach the goal you can combine the second or third category to avoid infinite loop. The “Stagnation Reached” category allows you to track the progress of fitness and terminate the process if the best fitness does not progress at the desired rate. To set the rate, you enter the minimum acceptable fitness progress in percentage over the expected period (i.e. the number of generation).

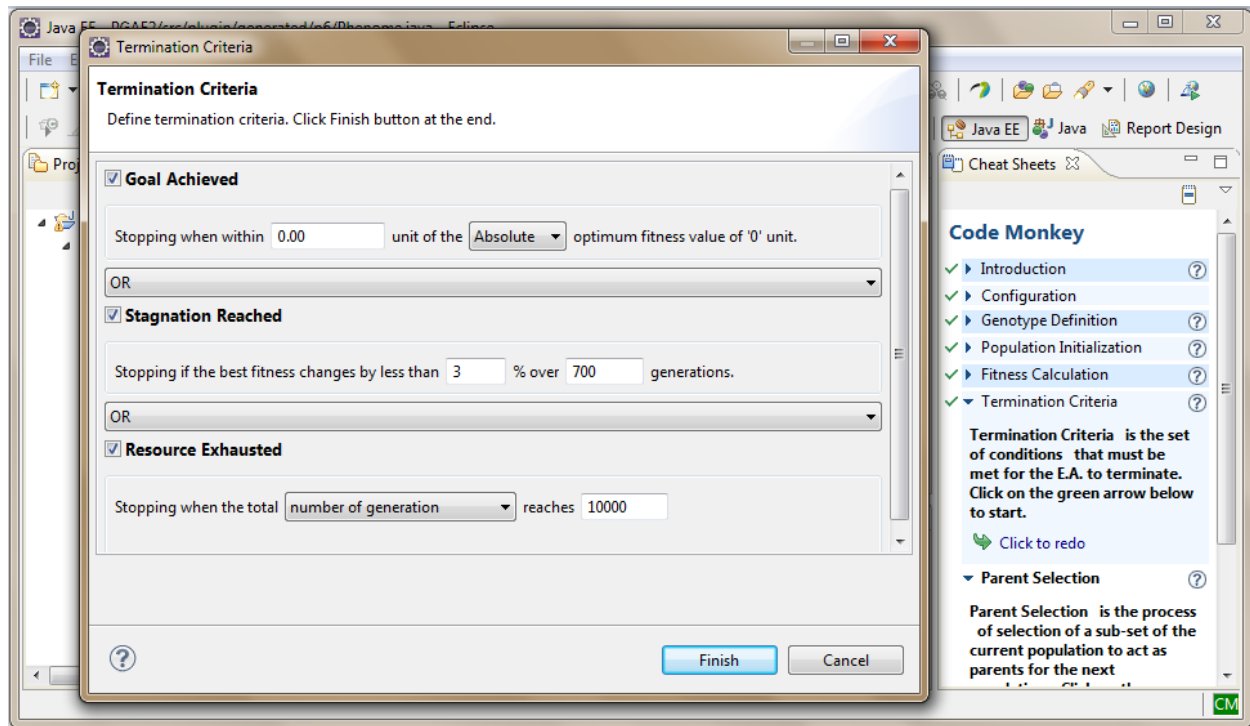


Figure B.47. Termination criteria wizard

The third category is “Resource Exhausted”. This is where you set a fix deadline for the EA process to stop regardless of fitness value. You have a choice of three options. You can specify a number and use it as maximum number of generation, execution time or number of fitness calculation.

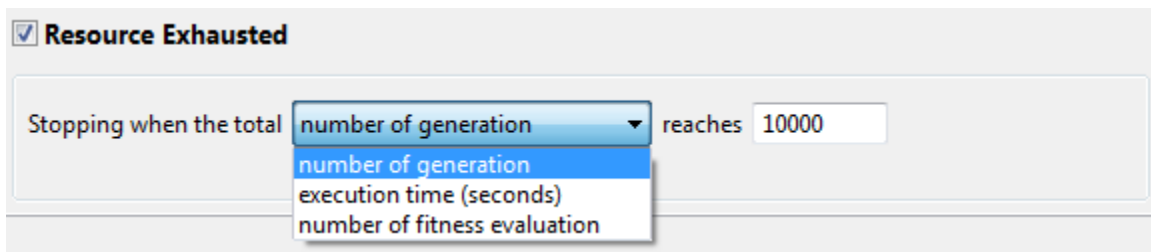


Figure B.48. Resource exhausted condition

Note that you can combine the categories with “AND” or “OR” options. After deciding on termination strategy, click on Finish button to generate the termination criteria code.

7. Parent Selection

Any EA process goes through a step of selecting individuals from current population and use them as parents to generate offsprings. In this step you provide the information on how this selection should happen. In the parent selection wizard the current population acts as input. You can see its size as you provided in an earlier step.

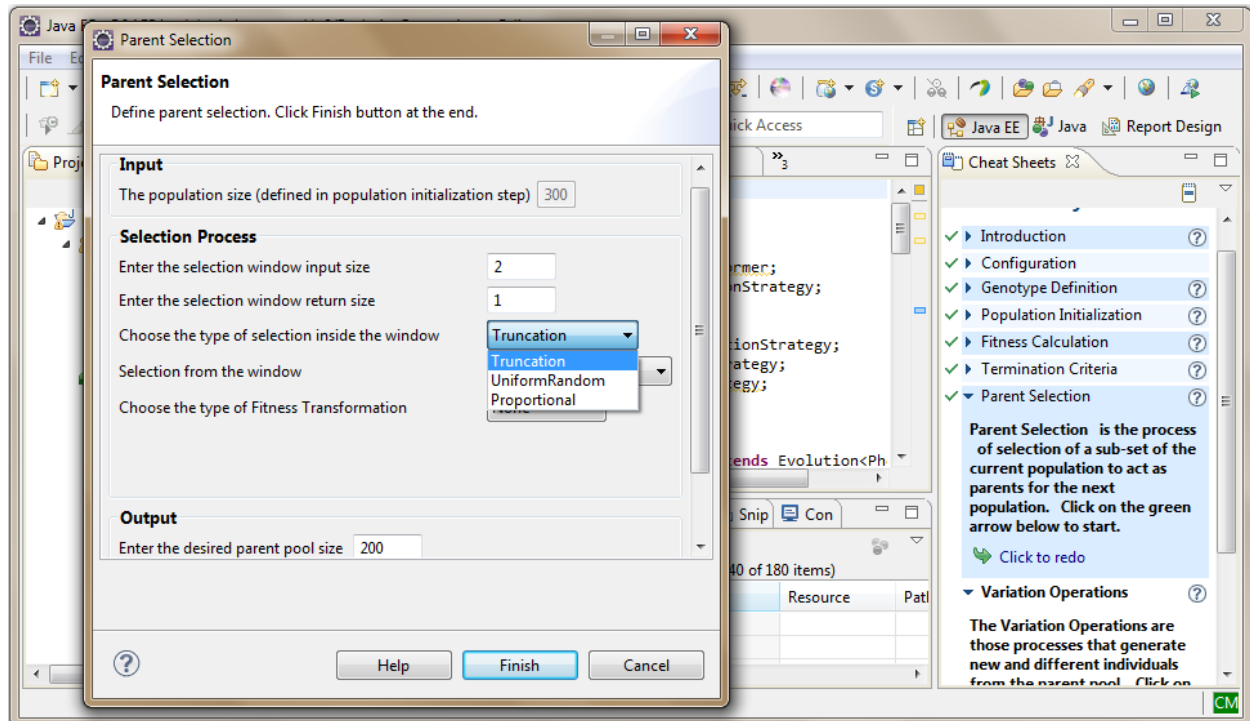


Figure B.49. Parent selection wizard

The selection process in CM is based on unification mechanism that is described in the Coke Monkey paper (http://link.springer.com/chapter/10.1007%2F978-3-642-37192-9_44). First you enter the size of a window that will be created from current population (it can be any number between 2 and population size). Then you define the size of window's output. Note that this number can't be bigger than window size. The window is filled randomly from population but selection from window will be defined by you. There are three main techniques; "Truncation", "UniformRandom" and "Proportional". Truncation is equivalent of deterministic selection. In case of Truncation the top N fittest individuals (where N is window's output size) will be picked from the window in one shot. In case of "Uniform Random" an individual is picked randomly from the window regardless of its fitness. In the case of Proportional each

individual is given a selection probability based on their fitness and selection is probabilistic and it is biased toward fittest individuals.

Optionally you can select a fitness transformation scheme. This will be mainly useful for Proportional selection. There are four different types of transformation provided by CM. Linear, Logarithmic, PowerLaw and Rank. Each may requires extra parameters when you choose from the drop down list.

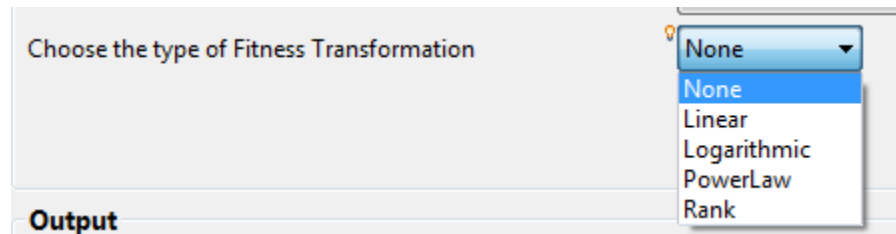


Figure B.50. Fitness transformation option

The Linear choice asks for two coefficients a & b for liner transformation formula: $tf(x) = af(x) + b$ where $f(x)$ is the fitness value before transformation and $tf(x)$ is transformed value.

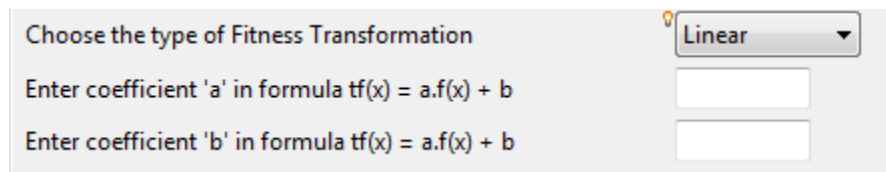


Figure B.51. Linear fitness transformation option

The Logarithmic transformation asks for one coefficient T for logarithmic formula: $tf(x) = e^{-f(x)/T}$

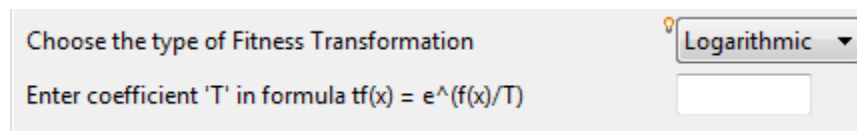
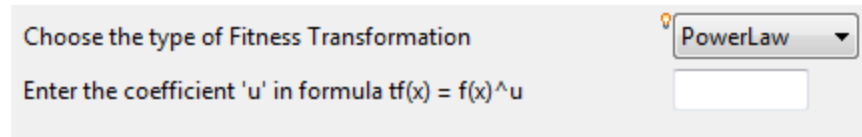


Figure B.52. Logarithmic fitness transformation option

The Power Law transformation asks for one coefficient u in the formula: $tf(x) = f(x)^u$



Choose the type of Fitness Transformation

Enter the coefficient 'u' in formula $tf(x) = f(x)^u$

PowerLaw

Figure B.53. Power law fitness transformation option

The Rank transformation does not require extra parameter because it calculates the rank of each individual based on their fitness and uses the rank number instead of the actual fitness value.

The other option related to selection process section is whether you want replacement (re-insertion) be allowed or not. If you choose “Without Replacement” the selection process guarantees that each individual is at maximum selected once as parent.

The last part of wizard is to set the size of parent pool which the output of this step. The selection process will calculate how many window it has to create and select from, to fill the parent pool. In case the numbers do not match you will be warned to modify the settings. For example if the ratio of window size over window output is very big and replacement is not allowed then the process may find that it is not able to create as many windows that is needed to fill the parent pool and it will warn you to change the numbers.

Once all data are provided and there is no warning you can click on Finish button to go to next step.

8. Variation Operations

After you decided how the parent pool is created you need to specify how the offsprings are created from the individuals in parent pool. This step will provide you with means to achieve that.

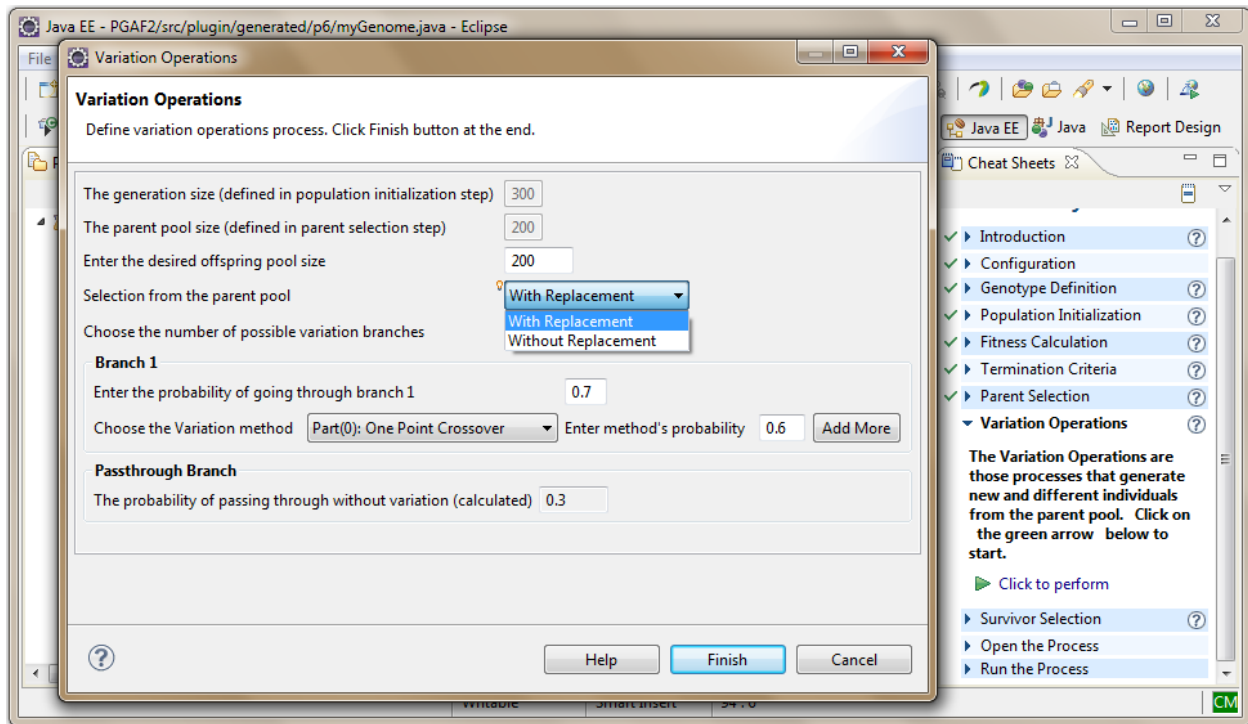


Figure B.54. Variation operations wizard

The first step in defining the variation strategy is to set the size of offspring pool where the result of variation operations goes into. To help you make that decision the population size and parent pool size that you defined earlier are shown. Then you need to decide if replacement is allowed in other words if each individual in parent pool gets maximum one chance to generate offspring (Without Replacement) or more (With Replacement).

Next you specify the number of parallel branches you need to apply your variation strategy. By default there are two branches, one to apply variation operations and one for pass-through without any variation. You can increase the number of branches up to the number of available operations.

All the variation operators that you selected in the Genotype Definition step are available here to choose. If for example in extreme case you want all operations to have their own path, you can do so. Remember that each branch requires a probability and the sum of all probabilities cannot exceed 1.0.

In each branch first you set the probability of that branch then from the list of available operations you choose the first method for that branch. You also provide a independent probability for that operation. If you want another operation happens after the first one you click on “Add More” button and select another

operation and subsequently assign an independent probability to it. You can continue adding more operations in each branch as many as you want to run sequentially with each operation having its own independent probability. Any added operation can be removed if you click on “Remove This” button except for the first one because it is required for each branch to have at least one operation.

Below is an example of two branches with the first branch having three sequential operations and the second branch having only one operation. Note that if you have created heterogeneous genotype there is a “part(n):” prefix for the operations that belong to genotype parts. This will help you to distinguish which operations in the list is for the subpart and which is for the whole genotype.

Branch 1

Enter the probability of going through branch 1

Choose the Variation method Enter its probability

Choose the Variation method Enter its probability

Choose the Variation method Enter its probability

Branch 2

Enter the probability of going through branch 2

Choose the Variation method Enter its probability

Passthrough Branch

The probability of passing through without variation (calculated)

Figure B.55. Variation operation branches definition

The probability of pass-through is calculated automatically. In above example no probability left for the pass-through branch.

Once you entered all the parallel and serial branches with their probabilities and desired operations you can click on Finish button. The CM application generates the code for the logic you provided and stores it in the variation class.

9. Survival Selection

This last step in an EA process cycle is creating the next generation also known as survival selection. This is where you specify how the current generation and offspring pool participate in creation of the next generation.

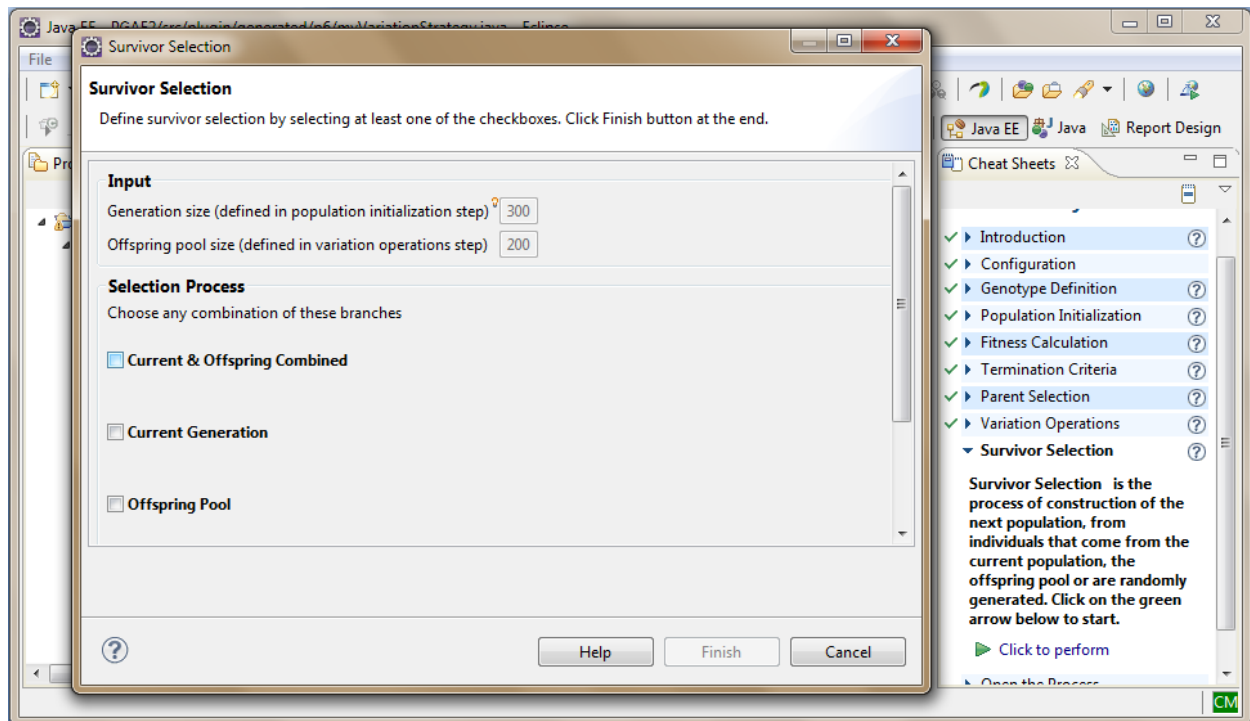


Figure B.56. Survival selection wizard

In this wizard page the size of both current population and offspring pool are shown based on what you defined earlier. You have a combination of five branches to choose from when deciding the survival selection logic as shown in the picture below.

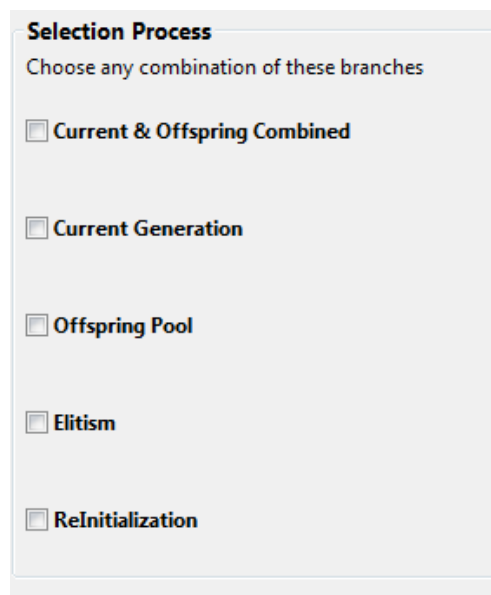
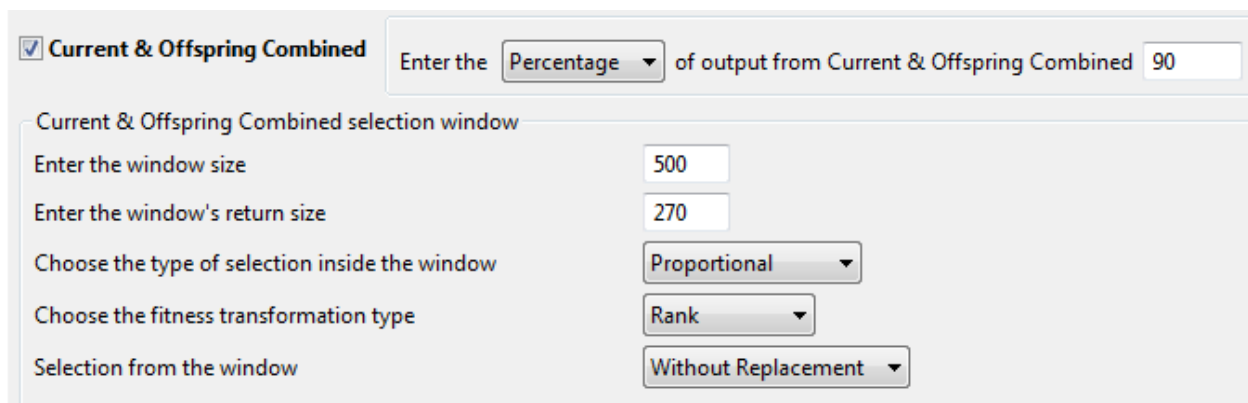


Figure B.57. Survival selection options

For any of branches that you select you need to specify what percentage of the next generation is created from that branch.

If you choose any of the first three branches, in addition to specifying their percentage, you will need to choose a selection mechanism similar to the one you defined for parent selection. The difference here is that the selected individuals go directly to the next generation.

For example in the picture below we chose the %90 of the next generation to be create from combination of current population and offspring pool based on proportional ranking selection without possibility of duplication.



The screenshot shows a software interface for configuring selection parameters. At the top, a checkbox labeled 'Current & Offspring Combined' is checked. To its right, a text field says 'Enter the' followed by a dropdown menu set to 'Percentage', then 'of output from Current & Offspring Combined' followed by a text field containing '90'. Below this is a section titled 'Current & Offspring Combined selection window'. Inside this section, there are five rows of configuration options: 'Enter the window size' with a text field containing '500'; 'Enter the window's return size' with a text field containing '270'; 'Choose the type of selection inside the window' with a dropdown menu set to 'Proportional'; 'Choose the fitness transformation type' with a dropdown menu set to 'Rank'; and 'Selection from the window' with a dropdown menu set to 'Without Replacement'.

Figure B.58. Survival selection detail

The last two branches (Elitism & Re-Initialization) are simpler and you will only need to specify their participation percentage. The Elitism branch is a way of preserving the best of current generation and passing them to the next generation. The “ReInitialization” branch on the other hand is a way of injecting new random individual to the next generation.

In the example below we chose %5 of the next generation to come from elitism and another %5 from Re-initialization.

The screenshot shows a configuration window with four options, each with a checkbox and a corresponding input field:

- ☐ **Current Generation**
- ☐ **Offspring Pool**
- ☒ **Elitism**: The input field contains "Enter the Percentage of output from Elitism 5".
- ☒ **ReInitialization**: The input field contains "Enter the Percentage of output from ReInitialization 5".

Figure B.59. Survival selection elitism and re-initialization options

The sum of all percentages must reach 100%. The CM application validated the numbers and warns you if there is any problem. Once you selected branches and fulfilled their parameters you can click on Finish button and the code will be generated for survival selection step in a new class in the package.

10. Open the Process

At this point the CM application has generated all required code for you. Assuming that you have entered the fitness evaluation code manually as it was requested in the Fitness Calculation step you are almost ready to launch the generated code. Once you click on “Open the Process” step in the guided navigator, the CM opens the entry class to your newly generated package in the eclipse editor. This is more for your information in case you want to view the generated codes. There is no need to modify anything. You can proceed to the next and final step in the wizard.

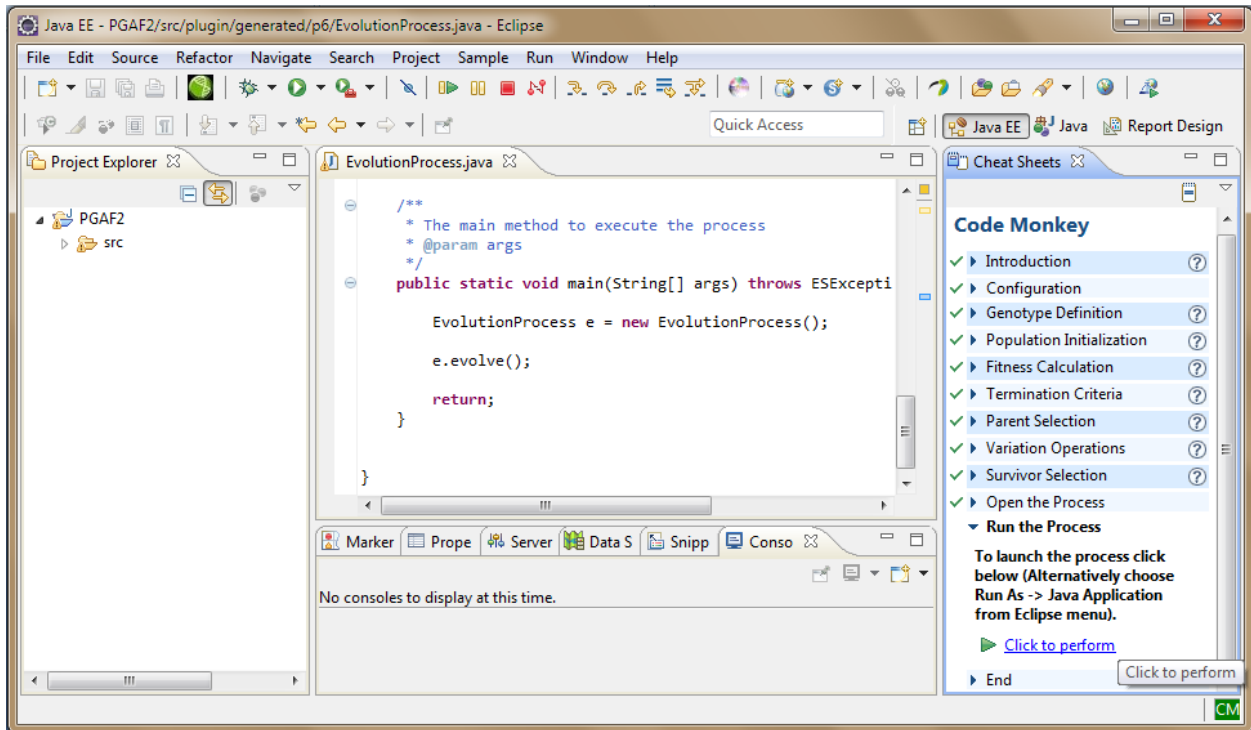


Figure B.60. CM entry point to generated code

11. Run the Process

The last step is to run the generated code. Once you hit the “Click to Perform” link in the guided navigator the main method in the opened file will be called and executed. The runtime output will be visible in the console. If there is any issue with executing the code you can alternatively right click on the opened file and choose “Run As” Java application from the context menu.

During the run two log files will be generated in the same folder at CM framework. One is similar to what is being logged on console. The other is a comma separated value (CSV) file that is ideal to be views and processed in an spreadsheet such as Microsoft Excel. You can use it to generate charts of mean fitness, best fitness and diversity over the course of execution. If they are not shown in the project explorer just right click on project explorer and choose refresh from the context menu.

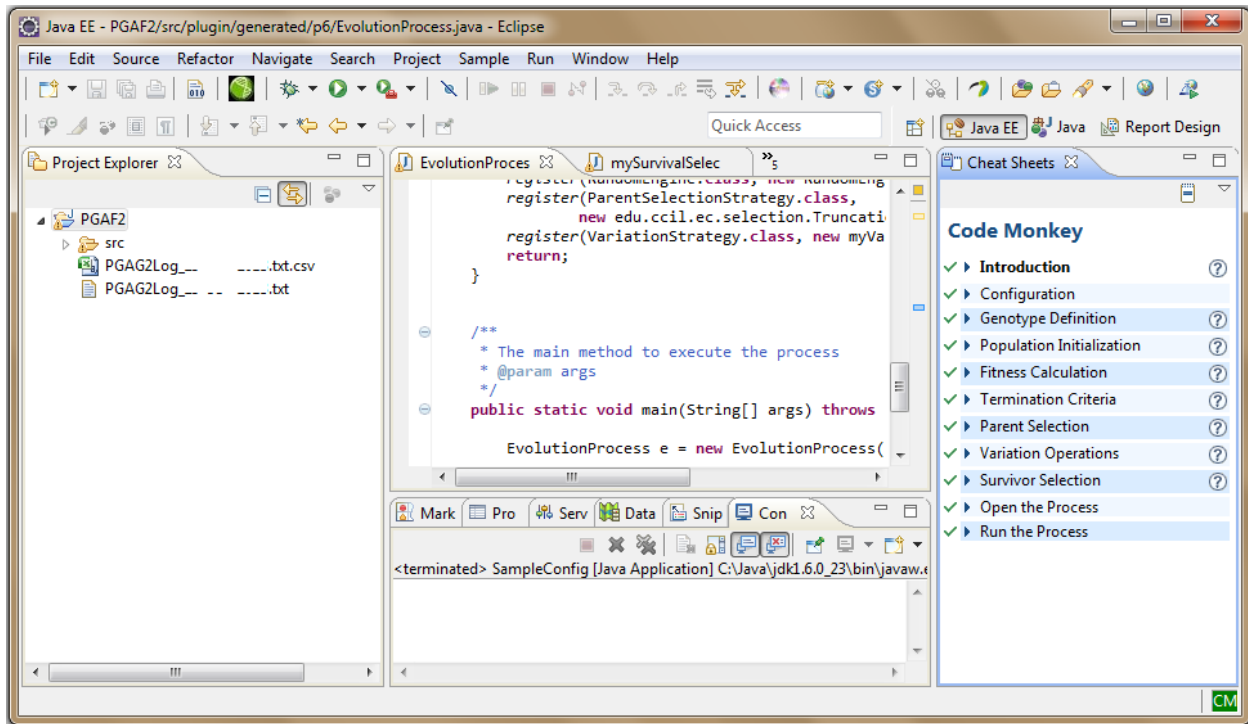


Figure B.61. Running the generated code

This concludes the Code Monkey user guide. You can get more information on each step of the CM application by clicking on question mark (?) icon next to each step. This will open the Eclipse content help with the page relate to that step. There will find more in-depth knowledge of inner working of CM framework itself.

Also if you want to change CM default preferences you can do so by going to Windows->Preferences in Eclipse and select the "CodeMonkey Preferences" from the left menu. Modify them at your own risk.