Library for Evolutionary Algorithm Programming (LEAP)

— Version 0.1.2 —

A live system to produce applications based on Evolutionary Computation

J.I. van Hemert
jvhemert@wi.leidenuniv.nl

Contents

1 Introduction

This documentation is far from complete. Just about half of it is done, but it still has to be checked. Before turning you over to the many pages of generated documentation some words and pictures to give you some overview and structure.

The most important thing to realize is that the library is build into two parts, an *abstract* part and an *implementation* part. To form a picture in your head look at Figure 1. The abstract part is the interface of the library, almost every class in the library only knows about other classes that are also in the abstract part. These classes do not have an implementation, or if they have, it will be very little. When a program is run it needs implemented code to get things done. This code is located in the implementation level. For now this implementation level is very thin, not much has been added to it, only some basics to get things going. When some class requires the implementation of another class this can be requested through the aid of factories. The factories decide which implementation is chosen to fill in a certain abstract class. This method of building is called an *Abstract Factory Method*.

2 Overview

Now let's turn to the next picture, which can be found in Figure 2. In there are the most important classes, the classes that make it an Evolutionary Algorithms library. To better understand the picture first some notational remarks:

Arrows with diamonds

With a closed diamond — The class with the diamond attached holds the class where the arrow points to and is responsible for its creation and destruction.

With an open diamond — The class with the diamond attached holds a reference pointer to the class where the arrow points to.

Borders of classes

Thick — The class holds a reference pointer to the class `Configuration`.

Striped — This class has no other class that holds it, the user of the library is responsible for creating and destroying it.

Some additional packages are there to support the library some more. There is an *Error* package (Figure 3) that implements all sorts of warnings and errors in the form of exceptions. These can be thrown and then catched and handled. A *Common* package (Figure 4) exists with some basic classes, such as strings and arrays.

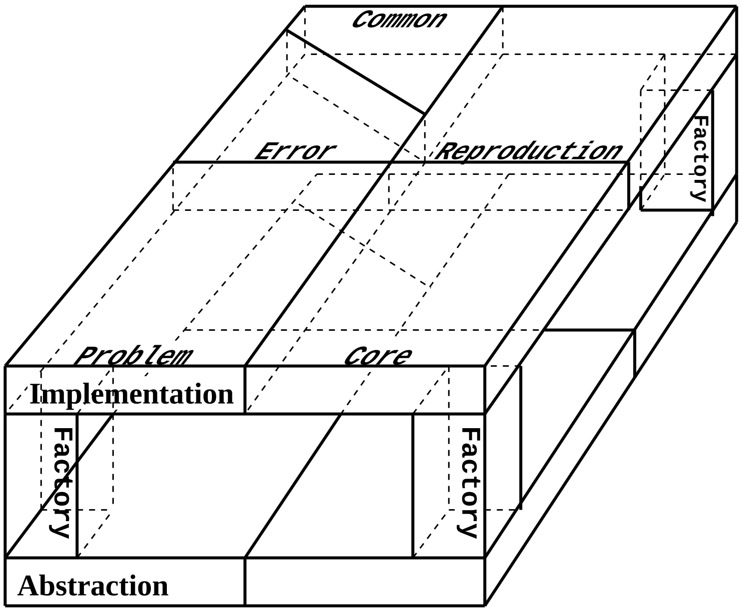


Figure 1: Overview of the library.

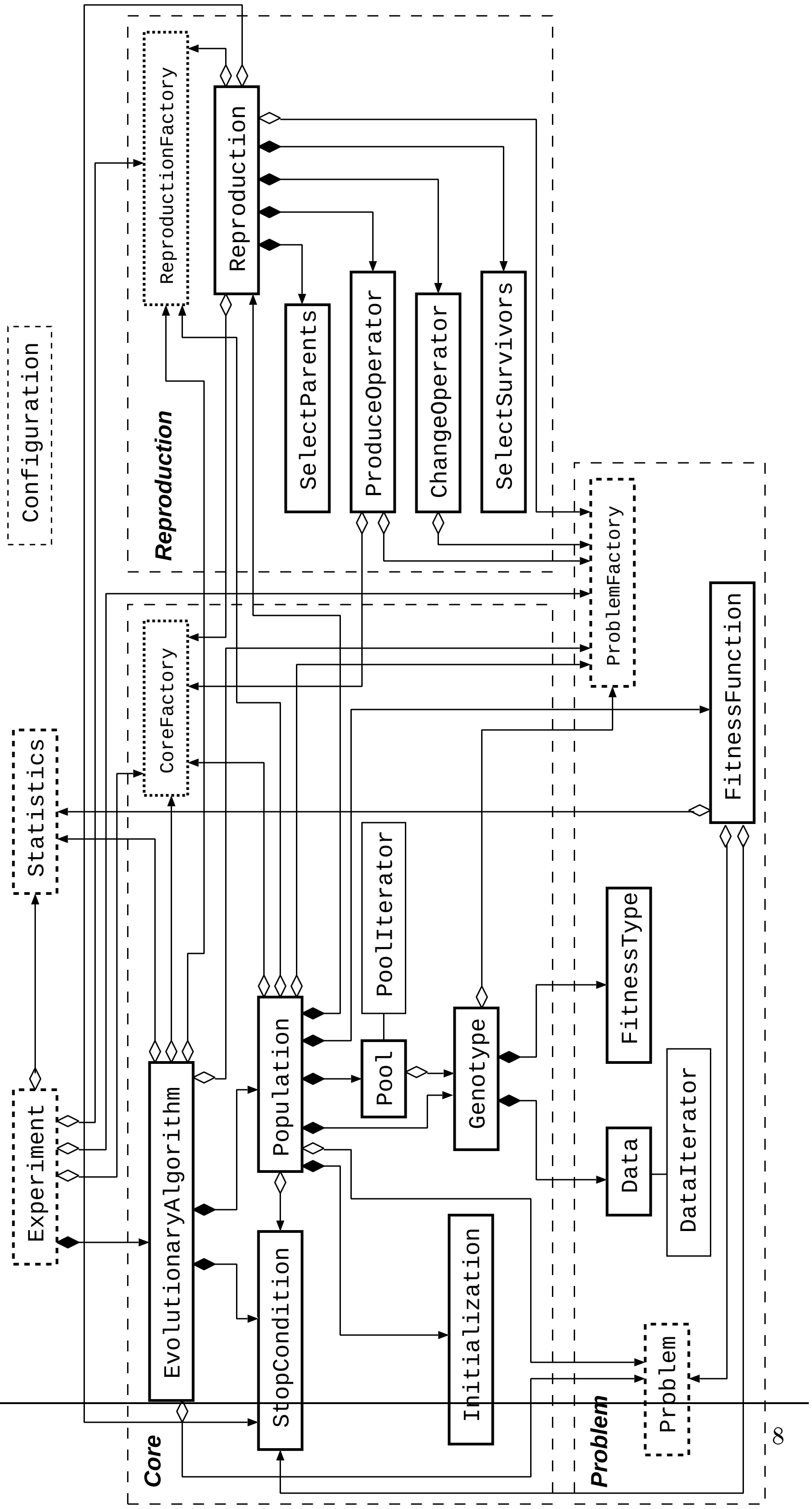


Figure 2: Most important classes and who holds/builds who.

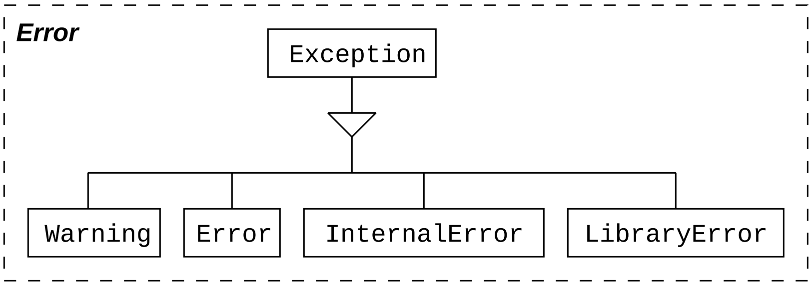


Figure 3: The *Error* package.

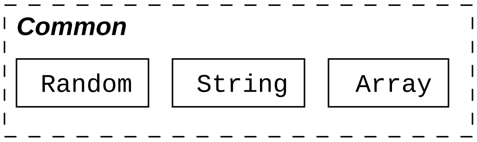


Figure 4: The *Common* package.

3

#define **Small error in DOC++**

Please note the small error DOC++ makes when generating documentation for an 'operator()' member function. The output results in '**operator)**'. Besides this error, somehow DOC++ sees this piece of writing as a #define statement.

4 Type definitions for CspC

General type definitons

Names

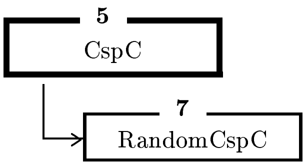
TreeC*	**Right**	*This container element's left and right brother*
TreeC*	**Parent**	*Its parent and first child*
typedef bool	**ConflictT**	*Conflict between the values of two variables.*
typedef unsigned int	**DomainSizeT**	*Size of the domain of a variable.*
typedef unsigned int	**NumberOfVariablesT**	*Number of variables in the CSP.*
typedef unsigned int	**VariableT**	*Value of a variable from the domain of the variable.*
typedef unsigned int	**ValueT**	*Value of a variable from the domain of the variable.*

These types can and should be used as a type for variables in your own code.

5 class **CspC**

Constraint Satisfaction Problem

Inheritance



Public Members

struct **ConstraintS** — *The storage for a constraint between two variables.*

typedef ConstraintS* **ConstraintT** — *The type for a constraint between two variables*

5.1 **CspC** (NumberOfVariablesT const iNumberOfVariables)
Constructor 15

5.2 **CspC** (const CspC & iCsp)
Copy constructor 15

5.3 CspC const& **operator=** (const CspC & iCsp)
Operator = ... 15

5.4 virtual **~CspC** () *Destructor* 16

5.5 NumberOfVariablesT const **GetNumberOfVariables** () const
Getting the number of variables 16

5.6 DomainSizeT const **GetDomainSize** (VariableT const iVariable) const
Getting the domainsize 17

5.7 void const **SetDomainSize** (VariableT const iVariable, DomainSizeT const iDomainSize)
Setting the domainsize 17

5.8 bool const **IsValid** (ValueT const iSolution []) const

Protected Members

Private Members

A base class for implementing Binary ConstrainS Satisfaction Problems.

5.1

CspC (NumberOfVariablesT const iNumberOfVariables)

Constructor

Constructor, allocates memory for all the objects except for the conflicts. The conflicts will be allocated as needed when constraints are added to the CSP.

Return Value:	nothing.
Parameters:	`iNumberOfVariables` — number of variables used in the CSP.
Author:	J.I. van Hemert
Version:	1.0

5.2

CspC (const CspC & iCsp)

Copy constructor

Copy constructor for CspC. Takes an existing CSP as input and copies it to the new CSP, including all constraints and conflicts.

Return Value:	nothing.
Parameters:	`iCsp` — existing object of CspC class.
Author:	J.I. van Hemert
Version:	1.0

5.3

CspC const& **operator=** (const CspC & iCsp)

Operator =

Destroys the left CSP and overwrites it with the right CSP. The left CSP will receive the constraints and conflicts of the right CSP.

Return Value: new CSP, copied from iCsp.
Parameters: `iCsp` — existing CSP.
Author: J.I. van Hemert
Version: 1.0

5.4

virtual **˜CspC** ()

Destructor

Destructor deallocates all memory used by the object.

Return Value: nothing.
Author: J.I. van Hemert
Version: 1.0

5.5

NumberOfVariablesT const **GetNumberOfVariables** () const

Getting the number of variables

Get the number of variables present in the CSP.

Return Value: number of variables in CSP.
Author: J.I. van Hemert
Version: 1.0

5.6

DomainSizeT const **GetDomainSize** (VariableT const iVariable) const

Getting the domainsize

Get the size of the domain of a variable.

Return Value: Size of the domain of the given variable.
Parameters: `iVariable` — A number corresponding to a variable.
Author: J.I. van Hemert
Version: 1.0

5.7

void const **SetDomainSize** (VariableT const iVariable, DomainSizeT const iDomainSize)

Setting the domainsize

Set the size of the domain of a variable.

Return Value: nothing.
Parameters: `iVariable` — A number corresponding to a variabe.
`iDomainSize` — size of the domain of iVariable.
Author: J.I. van Hemert
Version: 1.0

5.8

bool const **IsValid** (ValueT const iSolution []) const

Checking for valid solutions

A way of checking if solutions for the CSP are valid. The input array with the size of the number of variables in the problem is checked as a solution for the CSP. If there are no conflicts true is returned.

Return Value: true if the offered solution does not offend a conflict, false otherwise.
Parameters: `iSolution` — an array with values, where the first array index equals the first variable.
Author: J.I. van Hemert
Version: 1.0

5.9

int const **GetNumberOfConflicts** (ValueT const iSolution []) const

Getting the number of conflicts

A way of finding out how many conflicts a solution violates within the CSP. Given a array of size equal to the number of variables, check how many conflicts this solutions violates in the CSP.

Return Value: number of violated conflicts.
Parameters: `iSolution` — an array with values, where the first array index equals the first variable.
Author: J.I. van Hemert
Version: 1.0

5.10

unsigned int const **GetNumberOfConstraints** ( ) const

Getting the number of constraints

Get the number of constraints currently in the CSP.

Return Value: number of constraints in CSP.
Author: J.I. van Hemert
Version: 1.0

5.11

void const **AddConstraint** (VariableT const iVariable1, VariableT const iVariable2)

Adding constraints

Add a constraint between two variables.

Return Value: nothing.
Parameters: `iVariable1` — one of the two variables.
`iVariable2` — another variable.
Author: J.I. van Hemert
Version: 1.0

5.12

void const **RemoveConstraint** (VariableT const iVariable1, VariableT const iVariable2)

Removing constraints

Removes a constraint between two variables. Frees up memory from conflicts of the constraint.

Return Value: nothing.
Parameters: `iVariable1` — one of the two variables.
`iVariable2` — another variable.
Author: J.I. van Hemert
Version: 1.0

5.13

bool const **IsConstraint** (VariableT const iVariable1, VariableT const iVariable2) const

Testing constraints

Check for the existence of a constraint between two variables.

Return Value: true when the constraint exists, else it returns false.
Parameters: `iVariable1` — one of the two variables.
`iVariable2` — another variable.
Author: J.I. van Hemert
Version: 1.0

5.14

void const **AddConflict** (VariableT const iVariable1, VariableT const iVariable2, ValueT const iValue1, ValueT const iValue2)

Adding conflicts

Add a conflict between the values of two variables.

Return Value: nothing.
Parameters: `iVariable1` — one of the two variables.
`iVariable2` — another variable.
`iValue1` — value of first variable.
`iValue2` — value of the second variable.
Author: J.I. van Hemert
Version: 1.0

5.15

void const **RemoveConflict** (VariableT const iVariable1, VariableT const iVariable2, ValueT const iValue1, ValueT const iValue2)

Removing conflicts

Remove a conflict between the values of two variables.

Return Value: nothing.
Parameters: `iVariable1` — one of the two variables.
`iVariable2` — another variable.
`iValue1` — value of first variable.
`iValue2` — value of the second variable.
Author: J.I. van Hemert
Version: 1.0

5.16

bool const **IsConflict** (VariableT const iVariable1, VariableT const iVariable2, ValueT const iValue1, ValueT const iValue2) const

Testing conflicts

Check for an existing conflict between the values of two variables.

Return Value: true for existing constraint with conflict between values, false for non-existing conflict or non-existing constraint.

Parameters:
`iVariable1` — one of the two variables.
`iVariable2` — another variable.
`iValue1` — value of first variable.
`iValue2` — value of the second variable.

Author: J.I. van Hemert
Version: 1.0

5.17

bool const **SaveToFile** (char const * const iFilename) const

Saving a CSP

Save the CSP to a file. The fileformat is simple, first the number of variables is saved, then on the next line all the domainsizes of the variables are saved with spaces between them. A blank line follows, after which the conflicts are saved.

Return Value: true for success, false for a failure.
Parameters: `iFilename` — the filename of the file to save to.
Author: J.I. van Hemert

Version: 1.0

5.18

bool const **LoadFromFile** (char const * iFilename)

Loading a CSP

Load the CSP from a file.

Return Value: true for success, false for a failure.
Parameters: `iFilename` — the filename of the file to load from.
Author: J.I. van Hemert
Version: 1.0

5.19

bool const **IsDirty** () const

Testing dirty

Check if a CSP file has been touched.

Return Value: false if it is still empty, true if it has been touched.
Author: J.I. van Hemert
Version: 1.0

5.20

void const **Reset** ()

Resetting the CSP

Removes all constraints, conflicts and domainsizes from the CSP, freeing memory for the conflicts, but without freeing memory for the constraints. The CSP can be reused for a new CSP, because the number of variables is not cleared.

Return Value: nothing.
Author: J.I. van Hemert
Version: 1.0

5.21

virtual void const **ProduceProblem** ()

Virtual member-function

This *pure virtual function* has to be implemented by the class that is inherits from this class. It should produce a Constraint Satisfaction Problem according to some specific algorithm.

Return Value: nothing.

5.22

virtual void **ProduceFileName** (char * FileName)

Virtual member-function

This *pure virtual function* has to be implemented by the class that is inherits from this class. It should produce a filename that is unique for the parameters of problem made by the class it belongs to.

Return Value: nothing.

5.23

ConstraintT const **GetConstraint** (VariableT const iVariable1, VariableT const iVariable2) const

Getting constraints

Gets a constraint between two variables. Still not sure if we want this to be public or protected.

Return Value: a constraint.
Parameters: `iVariable1` — one of the two variables.
`iVariable2` — another variable.
See Also: the structure of a constraint in ConstraintS.
Author: J.I. van Hemert
Version: 1.0

5.24

bool const **IsValidParameter** (VariableT const iVariable) const

Testing variables

Check for the existence of a variable.

Return Value: true for existence, false for non-existence.
Parameters: `iVariable` — the variable to check.
Author: J.I. van Hemert
Version: 1.0

5.25

bool const **IsValidParameter** (VariableT const iVariable, ValueT const iValue) const

Testing values

Check for the existence of a variable and if its value lies in its domain.

Return Value: true for existence and valid value, false for non-existence or non-valid value.
Parameters: `iVariable` — the variable to check.
`iValue` — value of the variable.
Author: J.I. van Hemert
Version: 1.0

5.26

void const **Create** (NumberOfVariablesT const iNumberOfVariables)

Creating a CSP problem

Creates memory needed for holding the CSPs problem. initialises the CSP by allocating enough memory to hold all the constraints. No memory is allocated for the conflicts yet.

Return Value: nothing.
Parameters: `iNumberOfVariables` — number of variables used in the CSP.
Author: J.I. van Hemert
Version: 1.0

5.27

void const **Destroy** ()

Destroying the CSP

Deallocates memory from the CSP problem. Used in constructors and destructors.

Return Value: nothing.
Author: J.I. van Hemert
Version: 1.0

5.28

void const **Copy** (CspC const & iCsp)

Copying a CSP

Copies an existing CSP to this one by calling Create and copying all the constraints and conflicts.

Return Value: nothing.
Parameters: `iCsp` — existing CSP.
Author: J.I. van Hemert @version 1.0

5.29

void const **Swap** (VariableT & iVariable1, VariableT & iVariable2) const

Swap function

Swap the two parameters.

Return Value: nothing.
Parameters: `iVariable1` — First parameter.
`iVariable2` — Second parameter.
Author: J.I. van Hemert
Version: 1.0

6 Type definitions for RandomCspC

General type definitons

Names

typedef double
TightnessT *Determines the number of conflicts a constraint contains.*

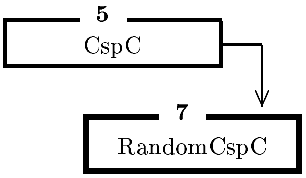
typedef double
ConnectivityT *Determines the number of nodes in the network.*

These types can and should be used as a type for variables in your own code.

7 class **RandomCspC** : public CspC

Random Constraint Satisfaction Problems

Inheritance



Public Members

This class is a empty class that can be extended to a class capable of producing Random Binary Constraint Satisfaction Problems. The class should implement the ProduceProblem function.

Author: J.I. van Hemert
Version: 1.1

7.1 **RandomCspC** ()

Constructor

Constructor copies all the parameters for further usage.

Return Value: nothing.

Parameters: `iNumberOfVariables` — Number of variables in CSP.
`iDomainSize` — Size of domain of each variable *or* maximum size of domain.
`iRandomDomainSize` — If true than a the domainsize will be varied over the variables.
`iTightness` — Determines percentage of conflicts that are generated.
`iConnectivity` — Percentage of constraints that are generated.

Author: J.I. van Hemert

Version: 1.1

7.2

˜RandomCspC ()

Destructor

For C++'s sake we have to implement the virtual destructor.

Return Value: nothing.

Author: J.I. van Hemert

Version: 1.1

7.3

void **ProduceFileName** (char * FileName)

Producing a filename

Create a filename from all the parameters of the CSP.

Return Value: nothing (FileName is the returnvalue).
Parameters: `FileName` — reference to a string (char *) that should have enough memory for holding the filename produced by this function, maximum required is 100 bytes.
Author: J.I. van Hemert
Version: 1.1

7.4

virtual void const **ProduceProblem** ()

Producing a Random CSP

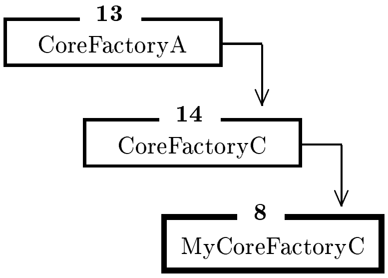
Produce a constraint satisfaction method using the method of Patrick Prosser, which he mentiones in his article 'Binary Constraint Satisfaction Problems: Some are Harder than Others' in '11th European Conference on Artificial Inteligence (ECAI94)' edited by A. Cohn and published in 1994 by * John Wiley & Sons, Ltd.

Return Value: nothing.
Author: J.I. van Hemert
Version: 1.1

8 class **MyCoreFactoryC** : public CoreFactoryC

Default factory of the Core package

Inheritance



Public Members

This is the default factory for the Core package. When it is used in a program, all the simple examples in the library will be created.

See Also: CoreFactoryA
Author: J.I. van Hemert
Version: 0.1

8.1 **MyCoreFactoryC** (ConfigurationA const & iConfiguration, RandomA const & iRandom)

Constructing a new factory for the Core package

```
CoreFactoryA * Corefactory

     = new MyCoreFactoryC(Configuration);
```

Return Value: Nothing.
Parameters: `iConfiguration` — A reference copy of configuration will be made and it will be distributed to every object made by the factory.
See Also: CoreFactoryA (→13.1, *page 49*)
Author: J.I. van Hemert
Version: 0.1

8.2 Create functions

Creating default objects in the Core package

Names

InitializationA*
CreateInitialization (ProblemA const & iProblem) const

See the abstract base class for further information.

See Also: CoreFactoryA
Author: J.I. van Hemert
Version: 0.1

9 template <class T> class **ArrayC**

A general array-class

Public Members

This template array-class has the advantage of checking numerous things, and giving sensible errors when something is going wrong. Furthermore it supports the naming of arrays, so you can have a unique name for every array. When an error occurs, you know instantly which array is the troublemaker. As most of the member functions behaviour is obvious, only some of them have additional explanation.

Author: J.I. van Hemert
Version: 0.1

9.1 Constructors

Names

9.1.1

ArrayC (unsigned int iSize)

Creating a new array

This is the wo/name constructor. When using this constructor errors will be reported without a name.

Return Value: Nothing.
Parameters: `iSize` — Size of the array.
Author: J.I. van Hemert
Version: 0.1

9.1.2

ArrayC (unsigned int iSize, StringC const & iString)

Creating a new array

This is the w/name constructor. When using this constructor errors will be reported witha name that identifies the array.

Return Value: Nothing.
Parameters: `iSize` — Size of the array.
`iName` — A useful description of the array
Author: J.I. van Hemert
Version: 0.1

9.1.3

ArrayC (ArrayC<T> const & iArray)

Copying an array

Copy constructor, destroys this array and copies the given one into it.

Return Value:	Nothing.
Parameters:	`iArray` — Another array of this class.
Author:	J.I. van Hemert
Version:	0.1

10 class **RandomA**

A randomgenerator

Public Members

Protected Members

This interface for a random-generator is included because of the fixed datatype used in the library. It has the basic functions of a random-generator, such as setting the seed and getting numbers within a range.

Author: J.I. van Hemert
Version: 0.1

10.1 virtual void **SetSeed** (DataType const & iSeed)

Providing a seed

This function sets the seeds and then reseeds the generator.

Return Value: Nothing.
Parameters: `iSeed` — The seed for the random-generator.
Author: J.I. van Hemert
Version: 0.1

10.2

virtual **operator DataType** ()

Getting a random number

Example:

```
void ShowRandomNumber(RandomA const & random)
{
    unsigned int r = random;
    cout << r << endl;
}
```

This function gets a random number. The range of this number depends on the datatype defined in the library.

Return Value: A random number.
Author: J.I. van Hemert
Version: 0.1

10.3

virtual DataType **operator)** (DataType iRange) const

Getting a random number in a range

Example:

```
void ShowRandomNumberUnderTen(RandomA const & random)
{
     unsigned int r = random(10);
     cout << r << endl;
}
```

This function gets a random number within the range zero to the number provided.

Return Value: Random number.
Parameters: `iRange` — Size of range where the random number should be from, range will be $0 \ldots (\texttt{iRange} - 1)$.
Author: J.I. van Hemert
Version: 0.1

10.4

virtual void **DoSeed** (DataType const & iSeed)

Initializing the random-generator

Provided with a seed, this function should initialize the random-generator.

Return Value: Nothing.
Parameters: `iSeed` — Seed used for initializing the random-generator.
Author: J.I. van Hemert
Version: 0.1

11 class **StringC**

Simple string handling

Public Members

Protected Members

Private Members

char* **String** *Datatype for storing the actual string.*

StringC implements simple string handling, like copying and concatenating. It has some extra features, such as casting to other types and input from and output to streams.

Author: J.I. van Hemert
Version: 0.1

11.1

Constructors.

Making new strings

Names

StringC () *Standard constructor.*

StringC (char const * const iChar)
string(char)*

StringC (int iInt)
string(int)

StringC (StringC const & iString)
string(string)

Examples:

```
StringC s("Test");

StringC t(s);

StringC r(24);

StringC i;

StringC n = "Str";

StringC g = i;
```

Return Value: Nothing.
Parameters: `iChar` — Char* to be copied.
`iInt` — Integer to be copied.
`iString` — String to be copied.
Author: J.I. van Hemert
Version: 0.1

11.2 Copy operators

*Copying strings and char*s*

Names

StringC& **operator=** (char const * const iChar)
*string = char**

StringC& **operator=** (StringC const & iString)
string = string

Examples:

```
StringC s;

StringC t;

s = "a string";

t = s;
```

Return Value: New string.
Parameters: `iChar` — Char* to be copied.
`iString` — String to be copied.
Author: J.I. van Hemert
Version: 0.1

11.3 Comparison operators

Comparing strings

Names

bool const **operator==** (StringC const & iString) const
string == string

bool const **operator==** (char const * const iChar) const
*string == char**

friend bool const
operator== (char const * const iChar,
StringC const & iString)
char == string*

bool const **operator!=** (char const * const iChar) const
*string != char**

bool const **operator!=** (StringC const & iString) const
string != string

friend bool const
operator!= (char const * const iChar,
StringC const & iString)
char != string*

Example:

```
StringC s("Yes");

if (s != "No");
        cout << "Equal";
```

Return Value: When strings are equal: `true` for operator== and `false` for operator!=.

Parameters: `iChar` — A constant string like "this is a string".
`iString` — A string of this class.

Author: J.I. van Hemert

Version: 0.1

11.4 Concatenation operators

*Concatenating strings and char*s*

Names

friend StringC
operator+ (char const * const iChar, StringC const & iString)
char + string*

StringC **operator+** (StringC const iString)
*string + char**

StringC& **operator+=** (char const * const iChar)
*string += char**

StringC& **operator+=** (StringC const iString)
string += string

Example:

```
StringC s("str");

StringC t("ing");

s += s + t;
```

Return Value: String concatenated with input.
Parameters: `iChar` — Char* to be concatenated.
`iString` — String to be concatenated.
Author: J.I. van Hemert
Version: 0.1

11.5 Stream operators

Getting and putting strings

Names

friend ostream&
operator<< (ostream & iStream, const StringC & iString)
stream << string

friend istream&

operator>> (istream & iStream,
StringC & iString)
stream >> string

Example:

```
StringC s("Something");
cout << s;
```

Return Value: Stream after handling string.
Parameters: `iStream` — Stream to work on.
`iString` — String to be printed/read.
Author: J.I. van Hemert
Version: 0.1

11.6 Overload operators

Changing strings to other types

Names

operator double const () const
double = string

operator char const * () const
char = string*

Example:

```
StringC s;
double d;
d = s;
```

Return Value: String as requested type.
Author: J.I. van Hemert
Version: 0.1

11.7

virtual void const **PrintToStream** (ostream & iStream) const

Implementation that formats the output of the string that is printed to a stream

Implementation that formats the output of the string that is printed to a stream.

Return Value: Nothing.
Parameters: `iStream` — Stream to output string on.
See Also: friend ostream & operator<<
Author: J.I. van Hemert
Version: 0.1

12 class **TreeA**

An n-ary tree.

This interface for an n-ary tree.

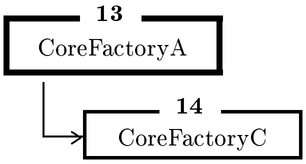
Author: J.I. van Hemert
Version: 0.1.2

13

class **CoreFactoryA**

Creating objects within the Core *package*

Inheritance



Public Members

This factory is used to spawn new objects in the *Core* package of the library. When it is inherited the Create memberfunctions will decide which class to use for each different object that can be spawned. Thus by changing the factory all the objects in the ibrary will use other objects.

Author: J.I. van Hemert
Version: 0.1

13.1

CoreFactoryA (ConfigurationA const & iConfiguration, RandomA const & iRandom)

Creating a factory for the Core *package*

Arguments

TreeElementC*
Right — *This container element's left and right brother*

TreeElementC*
Parent — *Its parent and first child*

This factory is inherited and the child determines for every Create member which class is used to spawn an object.

Return Value: Nothing.
Parameters: `iConfiguration` — The configuration used to determine settings for the algorithm and to provide a configuration to spawned objects within this class.
`iRandom` — A reference to this random object is stored and passed to classes who need it.
Author: J.I. van Hemert
Version: 0.1

13.2 Create functions

Creating objects in the Core *package*

Names

virtual EvolutionaryAlgorithmA*

CreateEvolutionaryAlgorithm (StatisticsA & iStatistics, ProblemFactoryA const & iProblemFactory, ReproductionFactoryA const & iReproductionFactory, ProblemA const & iProblem) const
Creating an EvolutionaryAlgorithm

virtual PopulationA*
CreatePopulation (StatisticsA & iStatistics, ProblemFactoryA const & iProblemFactory, ReproductionFactoryA const & iReproductionFactory, StopConditionA const & iStopCondition, ProblemA const & iProblem) const
Creating a Population

virtual GenotypeA*
CreateGenotype (ProblemFactoryA const & iProblemFactory) const
Creating a Genotype

virtual StopConditionA*
CreateStopCondition () const
Creating a StopCondition

virtual InitializationA*
CreateInitialization (ProblemA const & iProblem) const
Creating an Initialization

virtual PoolA*
CreatePool () const
Creating a Pool

Examples:

```
GenotypeA * Genotype =
    CoreFactory.CreateGenotype(ProblemFactory);
PoolA * Pool =
    CoreFactory.CreatePool();
```

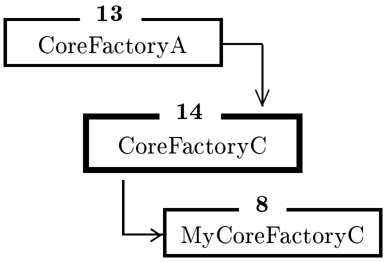
Parameters depend on the needs of the constructor of the object that is being created. See the appropriate class for details.

Return Value:	An object of a class inherited from the appropriate abstract class.
Author:	J.I. van Hemert
Version:	0.1

14 class **CoreFactoryC** : public CoreFactoryA

Default factory of the Core package

Inheritance



Public Members

This is the default factory for the Core package. When it is used in a program, all the simple examples in the library will be created.

See Also: CoreFactoryA
Author: J.I. van Hemert
Version: 0.1

14.1 **CoreFactoryC** (ConfigurationA const & iConfiguration, RandomA const & iRandom)

Constructing a new factory for the Core package

```
CoreFactoryA * Corefactory
     = new CoreFactoryC(Configuration);
```

Return Value: Nothing.
Parameters: `iConfiguration` — A reference copy of configuration will be made and it will be distributed to every object made by the factory.
See Also: CoreFactoryA (→13.1, *page 49*)
Author: J.I. van Hemert
Version: 0.1

14.2 Create functions

Creating default objects in the Core package

Names

virtual EvolutionaryAlgorithmA*
CreateEvolutionaryAlgorithm (StatisticsA & iStatistics, ProblemFactoryA const & iProblemFactory, ReproductionFactoryA const & iReproductionFactory, ProblemA const & iProblem) const

virtual PopulationA*

CreatePopulation (StatisticsA & iStatistics, ProblemFactoryA const & iProblemFactory, ReproductionFactoryA const & iReproductionFactory, StopConditionA const & iStopCondition, ProblemA const & iProblem) const

virtual GenotypeA*
CreateGenotype (ProblemFactoryA const & iProblemFactory) const

virtual StopConditionA*
CreateStopCondition () const

virtual InitializationA*
CreateInitialization (ProblemA const & iProblem) const

virtual PoolA*
CreatePool () const

See the abstract base class for further information.

See Also: CoreFactoryA
Author: J.I. van Hemert
Version: 0.1

15 class **EvolutionaryAlgorithmA**

Abstract Evolutionary Algorithm

Public Members

15.1 **EvolutionaryAlgorithmA** (ConfigurationA const & iConfiguration, StatisticsA & iStatistics, CoreFactoryA const & iCoreFactory, ProblemFactoryA const & iProblemFactory, ReproductionFactoryA const & iReproductionFactory, ProblemA const & iProblem)
Constructor for the abstract EvolutionaryAlgorithm class. 57

15.2 virtual void **Run** () *Abstract method for running the evolutionary algorithm* 58

This is the abstract class representing a basic Evolutionary Algorithm. It should run the loop that goes through all the generations.

Author: J.I. van Hemert
Version: 0.1

15.1

EvolutionaryAlgorithmA (ConfigurationA const & iConfiguration, StatisticsA & iStatistics, CoreFactoryA const & iCoreFactory, ProblemFactoryA const & iProblemFactory, ReproductionFactoryA const & iReproductionFactory, ProblemA const & iProblem)

Constructor for the abstract EvolutionaryAlgorithm class.

This constructor should only be called by its children. It copies the reference of the parameters into protected members, they can then be used by its children to spawn objects and read configuration parameters.

Return Value: Nothing.

Parameters:

`iConfiguration` — The configuration used to determine settings for the algorithm and to provide a configuration to spawned objects within this class.

`iStatistics` — The statistics are used to write data to and build a report after a run of the evolutionary algorithm.

`iCoreFactory` — The factory that is used to spawn objects from classes of the Core package, within this class. It is passed on to these spawned objects as well.

`iProblemFactory` — The factory that is used to spawn objects from classes of the Problem package, within this class. It is passed on to these spawned objects as well.

`iReproductionFactory` — The factory that is used to spawn objects from classes of the Reproduction package, within this class. It is passed on to these spawned objects as well.

`iProblem` — Actual problem the evolutionary algorithm has to work on.

Author: J.I. van Hemert

Version: 0.1

15.2

virtual void **Run** ()

Abstract method for running the evolutionary algorithm

Runs the evolutionary algorithm on the problem and writes statistics during and after the run.

Author: J.I. van Hemert
Version: 0.1

16 class **GenotypeA**

The genotype

Public Members

Protected Members

Private Members

static unsigned int **Unique_ID**	*Unique number generator*
unsigned int **ID**	*Genotype's unique identification number*

The genotype consists of a data and a fitness part. The data will be manipulated by the reproduction operators. The fitness part is used by the selection operators. Ofcourse additional ideas can be inorporated into children of this class.

Author: J.I. van Hemert
Version: 0.1

16.1

EvolutionaryAlgorithmC (ConfigurationA const & iConfiguration, StatisticsA & iStatistics, CoreFactoryA const & iCoreFactory, ProblemFactoryA const & iProblemFactory, ReproductionFactoryA const & iReproductionFactory, ProblemA const & iProblem)

This constructor is only called by the CoreFactory

Example:

```
EvolutionaryAlgorithmC
        EvolutionaryAlgorithm(Configuration, CoreFactory,
    ProblemFactory, ReproductionFactory, Problem);
```

This constructor is only called by the CoreFactory, and should *not* be called from somewhere else. If a new EvolutionaryAlgorithm class is to be used, the `CoreFactoryC` should be inherited and the `CreateEvolutionaryAlgorithm()` method should be overwritten. If more classes are changed it can be useful to inherit from `CoreFactoryA` class itself.

Return Value: Nothing.

Parameters:
`iConfiguration` — The configuration used to determine settings for the algorithm and to provide a configuration to spawned objects within this class.
`iStatistics` — The statistics are used to write data to and build a report after a run of the evolutionary algorithm.
`iCoreFactory` — The factory that is used to spawn objects from classes of the Core package, within this class. It is passed on to these spawned objects as well.
`iProblemFactory` — The factory that is used to spawn objects from classes of the Problem package, within this class. It is passed on to these spawned objects as well.
`iReproductionFactory` — The factory that is used to spawn objects from classes of the Reproduction package, within this class. It is passed on to these spawned objects as well.
`iProblem` — Actual problem the evolutionary algorithm has to work on.

See Also: CoreFactoryA
CoreFactoryC

Author: J.I. van Hemert

Version: 0.1

16.2

virtual void **Run** ()

Running the Evolutionary Algorithm

Example:

```
EvolutionaryAlgorithmA * EvolutionaryAlgorithm
        = CoreFactory->CreateEvolutionaryAlgorithm(ProblemFactory,
    ReproductionFactory);
EvolutionaryAlgorithm->Run();
```

It runs the evolutionary algorithm until the stopcondition is true and the it collects the data from the run.

Author: J.I. van Hemert
Version: 0.1

16.5

GenotypeA (ConfigurationA const & iConfiguration, ProblemFactoryA const & iProblemFactory)

Creating a new genotype

This constructor should only be called by its children. It copies the reference of the parameters into protected members, they can then be used by its children to spawn objects and read configuration parameters. Its children should create the following objects:

Data — which holds the data for reproduction

Fitness — used for selection purposes

Return Value: Nothing.
Parameters: `iConfiguration` — The configuration used to determine settings for the algorithm and to provide a configuration to spawned objects within this class.
`iProblemFactory` — The factory that is used to spawn objects from classes of the Problem package, within this class. It is passed on to these spawned objects as well.
Author: J.I. van Hemert
Version: 0.1

16.6

virtual void **Evaluate** (FitnessFunctionA const & iFitnessFunction)

Evaluating a genotype

This function evaluates the data of the genotype and saves it as its fitness. It does this using the provided fitnessfunction.

Return Value: Nothing.
Parameters: `iFitnessFunction` — Used for calculating the fitness of the data in this genotype.
Author: J.I. van Hemert
Version: 0.1

16.7

virtual void **Init** (InitializationA const & iInitialization)

Initializing a genotype

This function initialises the data of the genotype using the provided initialization unit.

Return Value: Nothing.
Parameters: `iInitialization` — Unit that fills the the data part with initial (random) data.
Author: J.I. van Hemert
Version: 0.1

16.8

virtual DataA& **GetData** () const

Looking/changing data

When data has to be changed/looked at by the reproduction unit, it can get a hold of it with this function.

Return Value: Data from genotype.
Author: J.I. van Hemert
Version: 0.1

16.9

virtual FitnessType const& **GetFitness** () const

Getting the fitnessvalue

Mostly for statistics usage this function gives a way to access the fitnessvalue of the genotype.

Return Value: Current fitnessvalue.
Author: J.I. van Hemert
Version: 0.1

16.10

unsigned int **GetUniqueID** () const

Getting a genotype's unique number

Every new created genotype gets a unique number, with this function this number can be viewed. Added for ease of debugging.

Return Value: Genotype's unique number.
Author: J.I. van Hemert
Version: 0.1

16.11

Comparator operators

Comparing genotypes

Names

bool	**operator==**	(GenotypeA const & iGenotype) const
bool	**operator!=**	(GenotypeA const & iGenotype) const
bool	**operator<**	(GenotypeA const & iGenotype) const
bool	**operator>**	(GenotypeA const & iGenotype) const
bool	**operator<=**	(GenotypeA const & iGenotype) const
bool	**operator>=**	(GenotypeA const & iGenotype) const

For comparing genotypes, or better said: for comparing genotype's fitness these operators are defined.

Return Value: Result of operation as a boolean.
Author: J.I. van Hemert
Version: 0.1

16.3 PopulationA* **Population**

This simple implementation has one population

This simple implementation has one population.

See Also: StopConditionA (→21, *page 75*)

16.4 StopConditionA* **StopCondition**

The stopcondition for requesting a termination of the run

The stopcondition for requesting a termination of the run.

See Also: PopulationA (→20, *page 71*)

17

class **InitializationA**

Initializing genotypes

With this class a genotype can initialize its data part. This will probably happen by a random fill of the data such that it does not violate the domain of the problem.

Author: J.I. van Hemert
Version: 0.1

18

class **PoolElementA**

Place inside the pool

Placeholder for a genotype. With the added feature of being able to hold a next and previous place.

Author: J.I. van Hemert
Version: 0.1

19 class **PoolA**

The pool

The pool is a container for holding the genotypes currently residing in the population. It's contents can be manipulated with the aid of iterators. There is not much to it then this, except that the pool keeps track of its current size. Because of the obvious names of all the members, no extra information is given for the seperate memberfunctions.

Author: J.I. van Hemert
Version: 0.1

20 class **PopulationA**

The population

Public Members

The population consists of a pool that holds the genomes and a reproduction unit that provides the population with a way to the next generation. In addtion to that it has a fitnessfunction for evaluating the genotypes in the pool.

Author: J.I. van Hemert
Version: 0.1

20.1

PopulationA (ConfigurationA const & iConfiguration, StatisticsA & iStatistics, CoreFactoryA const & iCoreFactory, ProblemFactoryA const & iProblemFactory, ReproductionFactoryA const & iReproductionFactory, StopConditionA const & iStopCondition, ProblemA const & iProblem)

Creating a new population

This constructor should only be called by its children. It copies the reference of the parameters into protected members, they can then be used by its children to spawn objects and read configuration parameters. Its children should create the following objects:

Initialization which is used for initialising genotypes

Reproduction needed to let the population evolve to the next generation

FitnessFunction needed to evaluate genotypes

Pool an empty pool where the genotypes reside

Return Value: Nothing.

Parameters:

`iConfiguration` — The configuration used to determine settings for the algorithm and to provide a configuration to spawned objects within this class.

`iStatistics` — The statistics are used to write data to and build a report after a run of the evolutionary algorithm.

`iCoreFactory` — The factory that is used to spawn objects from classes of the Core package, within this class. It is passed on to these spawned objects as well.

`iProblemFactory` — The factory that is used to spawn objects from classes of the Problem package, within this class. It is passed on to these spawned objects as well.

`iReproductionFactory` — The factory that is used to spawn objects from classes of the Reproduction package, within this class. It is passed on to these spawned objects as well.

`iProblem` — Actual problem the evolutionary algorithm has to work on.

Author: J.I. van Hemert
Version: 0.1

20.2

virtual void **Init** ()

Initializing the population

By calling this function the pool is filled with genotypes, these genotypes are initialised and evaluated before they are inserted into the pool.

Return Value: Nothing.
Author: J.I. van Hemert
Version: 0.1

20.3

virtual void **NextGeneration** ()

Evolving the population

This function should take the population to the next generation by using the Reproduction unit.

Return Value: Nothing.
Author: J.I. van Hemert
Version: 0.1

20.4

virtual GenotypeA const& **GetBestIndividual** () const

Getting the best individual

A function to get the best individual of the population.

Return Value:	The best genotype according to the fittnessfunction.
Author:	J.I. van Hemert
Version:	0.1

21 class **StopConditionA**

Terminating the evolutionary algorithm

Public Members

An object that gets this class as a parameter can request a termination of the evolutionary algorithm.

Author: J.I. van Hemert
Version: 0.1

21.1 virtual void **RequestStop** () const

Terminating the evolutionary algorithm

Examples:

```
if (Statistics.GetNumberOfEvaluations() ==
    MaximumAllowedNumberOfEvaluations)

     StopCondition.RequestStop();
```

Return Value: Nothing.
Author: J.I. van Hemert
Version: 0.1

21.2 virtual void **FoundSolution** () const

A solution has been found

Examples:

```
if (Genotype.GetFitness() == Optimum)

     StopCondition.FoundSolution();
```

Return Value: Nothing.
Author: J.I. van Hemert
Version: 0.1

21.3 Information functions

Checking state of stopconditon

Names

virtual bool
IsSolutionFound () const

virtual bool
IsStopRequested () const

Examples:

```
while (!StopCondition->IsStopRequested())
```

// Let the evolutionary algorithm run

```
if (StopCondition->IsSolutionFound());

    cout << "Found solution" << endl;
```

Return Value: Boolean value answering question.
Author: J.I. van Hemert
Version: 0.1

22 class **ExceptionA**

An exception

Public Members

Base class for all kinds of errors and warnings within the library. These exceptions will be created and then trown by objects, they should be catched, handled and neatly disposed of.

```
catch(ExceptionA* Exception)
{
    Exception->Handle();
    delete Exception;
}
```

Author: J.I. van Hemert
Version: 0.1

22.1 virtual void **Handle** () const

Handling an exception

This function should printout a newline and its name appended with a colon, call `PrintMessage()` and then do what is should do. For example in the case of a fatal error the following will be the case:

```
void FatalError::Handle()
{
```

```
    cerr << endl << "FatalError:  ";
    PrintMessage();
    exit(1);
}
```

Author: J.I. van Hemert
Version: 0.1

23 class **DataA**

A genotype's data

Public Members

This holds the data of a genotype. This data will be manipulated by the reproduction operators and will be evaluated by the fitnessfunction. It has iterators to hide the inner structure.

Author: J.I. van Hemert
Version: 0.1

23.1 class **DataConstIteratorA**

Viewing data

This iterator should be used when the data is to be viewed.

Author: J.I. van Hemert
Version: 0.1

23.2 class **DataIteratorA**

Changing data

This iterator should be used when the data is to be changed.

Author: J.I. van Hemert
Version: 0.1

23.3 Iterator creators

Creating iterators

Names

virtual DataIteratorA* **CreateDataIterator** ()

virtual DataIteratorA* **CreateDataIterator** (DataIteratorA const & iDataIterator)

virtual DataConstIteratorA* **CreateDataConstIterator** () const

virtual DataConstIteratorA* **CreateDataConstIterator** (DataConstIteratorA const & iDataConstIterator) const

These functions can be used to create iterators for traversing the data. Also provided are functions to copy existing iterators.

Author: J.I. van Hemert
Version: 0.1

24 class **FitnessFunctionA**

A function for awarding fitness to genotypes

Public Members

This fitnessfunction will be used by the genotype to evaluate its data and get a fitnessvalue. These fitnessvalues can then be used for ordening the genotypes in a population.

Author: J.I. van Hemert
Version: 0.1

24.1 **FitnessFunctionA** (ConfigurationA const & iConfiguration, StatisticsA & iStatistics, StopConditionA const & iStopCondition, ProblemA const & iProblem)

Creating a new fitnessfunction

This constructor should only be called by its children. It copies the reference of the parameters into protected members, they can then be used by its children to spawn objects and read configuration parameters.

Return Value: Nothing.
Parameters: `iConfiguration` — The configuration used to determine settings for the algorithm and to provide a configuration to spawned objects within this class.
`iStatistics` — The statistics are used to write data to and build a report after a run of the evolutionary algorithm.
`iProblem` — Actual problem the evolutionary algorithm has to work on, used for evaluating the data.
Author: J.I. van Hemert
Version: 0.1

24.2

virtual void const **Calculate** (FitnessType & iFitnessValue, DataA const & iData) const

Evaluating data

Example:

```
void Genotype::Evaluate(FitnessFunctionA const &
    FitnessFunction)
{
     FitnessFunction(GetFitness(), GetData());
}
```

The fitnessvalue provided is changed to reflect the fitness of the data that is provided.

Return Value: See iFitnessValue.
Parameters: `iFitnessValue` — This fitnessvalue is altered to reflect the data provided.
`iData` — This is the data that will be evaluated.
Author: J.I. van Hemert
Version: 0.1

25 class **ProblemFactoryA**

Creating objects within the Problem *package*

Public Members

This factory is used to spawn new objects in the *Problem* package of the library. When it is inherited the Create memberfunctions will decide which class to use for each different object that can be spawned. Thus by changing the factory all the objects in the ibrary will use other objects.

Author: J.I. van Hemert
Version: 0.1

25.1 **ProblemFactoryA** (ConfigurationA const & iConfiguration)

Creating a factory for the Problem *package*

This factory is inherited and the child determines for every Create member which class is used to spawn an object.

Return Value: Nothing.
Parameters: `iConfiguration` — The configuration used to determine settings for the algorithm and to provide a configuration to spawned objects within this class.
Author: J.I. van Hemert

Version: 0.1

25.2 Create functions

Creating objects in the Problem *package*

Names

virtual FitnessFunctionA*
CreateFitnessFunction (StopConditionA const & iStopCondition, StatisticsA & iStatistics, ProblemA const & iProblem) const
Creating a FitnessFunction

virtual DataA*
CreateData (unsigned int iSize) const
Creating a Data object

virtual ProblemA*
CreateProblem ( ) const
Creating a Problem object

Examples:

```
FitnessFunctionA * FitnessFunction =

     ProblemFactory.CreateFitnessFunction(StopCondition,
        Statistics, Problem);

DataA * Data =

 ProblemFactory.CreateData(100);
```

Parameters depend on the needs of the constructor of the object that is being created. See the appropriate class for details.

Return Value: An object of a class inherited from the appropriate abstract class.

Author: J.I. van Hemert

Version: 0.1

26

class **ChangeOperatorA**

The change operator

Public Members

This operator takes existing genotypes and changes them. No new genotypes are produced this way.

Author: J.I. van Hemert
Version: 0.1

26.1

ChangeOperatorA (ConfigurationA const & iConfiguration, ProblemFactoryA const & iProblemFactory, ProblemA const & iProblem, RandomA const & iRandom)

Creating a new change operator

This constructor should only be called by its children. It copies the reference of the parameters into protected members, they can then be used by its children to spawn objects and read configuration parameters.

Return Value: Nothing.
Parameters: `iConfiguration` — The configuration used to determine settings for the algorithm and to provide a configuration to spawned objects within this class.
`iProblemFactory` — The factory that is used to spawn objects from classes of the Problem package, within this class. It is passed on to these spawned objects as well.
`iRandom` — The production operator will require some random numbers.
Author: J.I. van Hemert
Version: 0.1

26.2

virtual void **operator)** (ArrayC<GenotypeA*> const & iArray) const

Changing offspring

```
ArrayC<GenotypeA*>* Children;
// Produce a list of children and store them in the Children list
ChangeOperator(*Children);
```

Given a list of genotypes, the genotypes are changed according to some algorithm.

Return Value: iArray is the returned variable.
Parameters: `iArray` — This is a list of to-be-changed genotypes.
Author: J.I. van Hemert
Version: 0.1

27 class **ProduceOperatorA**

The production operator

Public Members

This operator should produce new genotypes using the provided existing ones. The existing genotype are not changed. An other way of putting it: Given a number of parents this operator produces a number of children.

Author: J.I. van Hemert
Version: 0.1

27.1 **ProduceOperatorA** (ConfigurationA const & iConfiguration, CoreFactoryA const & iCoreFactoryA, ProblemFactoryA const & iProblemFactory, ProblemA const &iProblem, RandomA const & iRandom)

Creating a new production operator

This constructor should only be called by its children. It copies the reference of the parameters into protected members, they can then be used by its children to spawn objects and read configuration parameters.

Return Value:	Nothing.
Parameters:	`iConfiguration` — The configuration used to determine settings for the algorithm and to provide a configuration to spawned objects within this class. `iCoreFactory` — The factory that is used to spawn objects from classes of the Core package, within this class. It is passed on to these spawned objects as well. `iProblemFactory` — The factory that is used to spawn objects from classes of the Problem package, within this class. It is passed on to these spawned objects as well. `iRandom` — The production operator will require some random numbers.
Author:	J.I. van Hemert
Version:	0.1

27.2

virtual ArrayC <GenotypeA*> * **operator)** (ArrayC<GenotypeA const *> const & iArray)

Producing offspring

```
ArrayC<GenotypeA const *>* Parents;
ArrayC<GenotypeA*>* Children;
// Select parents and store them in the Parents list
Children = ProduceOperator(*Parents);
```

This is the only function of the production operator. It returns a number of newly generated children using the provided list of existing ones.

Return Value:	Newly generated genotypes.
Parameters:	`iArray` — This is a list of parent genotypes.
Author:	J.I. van Hemert
Version:	0.1

28

class **ReproductionA**

The population

Public Members

28.1 **ReproductionA** (ConfigurationA const & iConfiguration, CoreFactoryA const & iCoreFactory, ProblemFactoryA const & iProblemFactory, ReproductionFactoryA const & iReproductionFactory, StopConditionA const & iStopCondition, ProblemA const & iProblem, RandomA const & iRandom)
Creating a new population 92

28.2 virtual void **operator)** (PoolA & iPool, FitnessFunctionA const & iFitnessFunction, StatisticsA & iStatistics)
Evolving a population 93

This is the reproduction unit within the library. It is called by the population to evolve to the next generation. It makes a number of supporting objects to aid all the work that as to be done. This work depends on what kind of operators and selection mechanism the evolutionary algorithm will have to contain. Most evolutionary algorithms make use of these four kinds op operators and mechanisms:

SelectParents Selecting which genotypes will be used for reproducing purposes.

ProduceOperator Producing new genotypes

ChangeOperator Changing existing genotypes

SelectSurvivors Selecting who will be in the next generation

Author: J.I. van Hemert
Version: 0.1

28.1

ReproductionA (ConfigurationA const & iConfiguration, CoreFactoryA const & iCoreFactory, ProblemFactoryA const & iProblemFactory, ReproductionFactoryA const & iReproductionFactory, StopConditionA const & iStopCondition, ProblemA const & iProblem, RandomA const & iRandom)

Creating a new population

This constructor should only be called by its children. It copies the reference of the parameters into protected members, they can then be used by its children to spawn objects and read configuration parameters. Its children should create the following objects:

SelectParents — Selecting which genotypes will be used for reproducing purposes.

ProduceOperator — This operator takes a number of existing genotypes (parents) and creates a number of new ones (children, offspring). Examples of such operators are recomination- and crossover-operators.

ChangeOperator — This operator takes a number of existing genotypes and alters them, without creating new genotypes. Examples of such operators are mutation- and inversion-operators.

SelectSurvivors — When a population is changed and grown in size, often the population has to be brought back to the previous size. To do this a selection mechanism is needed to choose who stays and who will be killed.

Return Value: Nothing.

Parameters: `iConfiguration` — The configuration used to determine settings for the algorithm and to provide a configuration to spawned objects within this class.
`iCoreFactory` — The factory that is used to spawn objects from classes of the Core package, within this class. It is passed on to these spawned objects as well.
`iProblemFactory` — The factory that is used to spawn objects from classes of the Problem package, within this class. It is passed on to these spawned objects as well.
`iReproductionFactory` — The factory that is used to spawn objects from classes of the Reproduction package, within this class. It is passed on to these spawned objects as well.
`iStopCondition` — The reproduction unit has the right to terminate the evolutionary algorithm, for instance when the maximum number of operations allowed is reached.
`iRandom` — Random numbers for the operators.
Author: J.I. van Hemert
Version: 0.1

28.2

virtual void **operator)** (PoolA & iPool, FitnessFunctionA const & iFitnessFunction, StatisticsA & iStatistics)

Evolving a population

This is the only function of the reproduction unit. When called it will turn the operators and selection mechanins loose on the provided pool.

Return Value: Nothing.
Parameters: `iPool` — This is the pool of genotypes the reproduction unit will work on.
`iFitnessFunction` — Used to evaluate newly generated genotypes.
`iStatistics` — Here we can store some statistics, for instance the number of operations performed.
Author: J.I. van Hemert

Version: 0.1

29 class **ReproductionFactoryA**

Creating objects within the Reproduction *package*

Public Members

This factory is used to spawn new objects in the *Reproduction* package of the library. When it is inherited the Create memberfunctions will decide which class to use for each different object that can be spawned. Thus by changing the factory all the objects in the ibrary will use other objects.

Author: J.I. van Hemert
Version: 0.1

29.1 **ReproductionFactoryA** (ConfigurationA const & iConfiguration, RandomA const & iRandom)

Creating a factory for the Reproduction *package*

This factory is inherited and the child determines for every Create member which class is used to spawn an object.

Return Value: Nothing.
Parameters: `iConfiguration` — The configuration used to determine settings for the algorithm and to provide a configuration to spawned objects within this class.
`iRandom` — A reference to this random object is stored and passed to classes who need it.
Author: J.I. van Hemert
Version: 0.1

29.2 Create functions

Creating objects in the Reproduction *package*

Names

virtual ReproductionA*
CreateReproduction (CoreFactoryA const & iCoreFactory, ProblemFactoryA const & iProblemFactory, ProblemA const & iProblem, StopConditionA const & iStopCondition) const
Creating a Reproduction unit

virtual SelectParentsA*
CreateSelectParents () const
Creating a Parents selections mechanism

virtual SelectSurvivorsA*
CreateSelectSurvivors () const
Creating a Surviviors selections mechanism

virtual ChangeOperatorA*
CreateChangeOperator (ProblemFactoryA const & iProblemFactory, ProblemA const & iProblem) const
Creating a Change operator

virtual ProduceOperatorA*

CreateProduceOperator (CoreFactoryA const & iCoreFactory, ProblemFactoryA const & iProblemFactory, ProblemA const & iProblem) const

Creating a Produce operator

Examples:

```
SelectParentsA * SelectParents =
    ReproductionFactory.CreateSelectParents();
ChangeOperatorA * ChangeOperator =
    ReproductionFactory.CreateChangeOperator(ProblemFactory);
```

Parameters depend on the needs of the constructor of the object that is being created. See the appropriate class for details.

Return Value: An object of a class inherited from the appropriate abstract class.
Author: J.I. van Hemert
Version: 0.1

30 class **SelectParentsA**

The parents selection mechanism

Public Members

This selection mechanism is used for filling a list of genotypes. These genotypes act as parents for the production of new offspring.

See Also: ProduceOperator
Author: J.I. van Hemert
Version: 0.1

30.1 **SelectParentsA** (ConfigurationA const & iConfiguration, RandomA const & iRandom)

Creating a new parent selection mechanism

This constructor should only be called by its children. It copies the reference of the parameters into protected members, they can then be used by its children to spawn objects and read configuration parameters.

Return Value: Nothing.
Parameters: `iConfiguration` — The configuration used to determine settings for the algorithm and to provide a configuration to spawned objects within this class.
`iRandom` — The mechanis will often require some random numbers.

Author: J.I. van Hemert
Version: 0.1

30.2

virtual ArrayC <GenotypeA const *> * **operator)** (PoolA const & iPool) const

Getting a list of parents

```
ArrayC<GenotypeA const *>* Parents;

Parents = SelectParents(Pool);
```

Given a pool filled with genotypes, this function selects a list of these genotypes and returns them.

Return Value:	An array filled with pointers to genotypes in the pool.
Parameters:	`iPool` — A pool with genotypes.
Author:	J.I. van Hemert
Version:	0.1

31 class **SelectSurvivorsA**

The survivors selection mechanism

Public Members

This selection mechanism takes a pool and a list containing the offspring. It then merges these two to form the pool for the next generation.

Author: J.I. van Hemert
Version: 0.1

31.1 **SelectSurvivorsA** (ConfigurationA const & iConfiguration)

Creating a new survivors selection mechanism

This constructor should only be called by its children. It copies the reference of the parameters into protected members, they can then be used by its children to spawn objects and read configuration parameters.

Return Value: Nothing.
Parameters: `iConfiguration` — The configuration used to determine settings for the algorithm and to provide a configuration to spawned objects within this class.

Author: J.I. van Hemert
Version: 0.1

31.2

virtual void **operator)** (PoolA & iPool, ArrayC<GenotypeA*> const & iArray) const

Creating the pool for the next generation

```
ArrayC<GenotypeA*>* Children;
// Produce a list of children and store them in the Children list
SelectSurvivors(Pool, *Children);
```

Given a pool and a list of genotypes, the two are merged into a new pool which overwrites the old one.

Return Value: iPool is the returned variable.
Parameters: `iPool` — This is the pool of the current generation.
`iArray` — This is a list of offspring that was created.
Author: J.I. van Hemert
Version: 0.1

32 class **ConfigurationA**

Reading configuration files

Public Members

Protected Members

virtual void const
PrintToStream (ostream & iStream) const
Prints the loaded configuration to the stream.

virtual StringC const
GetStringItem (StringC const & iClass, StringC const & iItem) const throw(ExceptionA*)
This function retrieves an item given a class and a variable name.

The configuration class reads a file containing items looking like this: '`NumberOfRuns = 10`'. It also makes use of classes, these look like: '`[Experiment]`'. All items on lines below a line containing a class, and above a line containing some other class, belong to that first class. Using this items can be grouped together. For example:

```
[Experiment]

NumberOfRuns = 10
```

Means that '`NumberOfRuns`' belongs to '`Experiment`' and when we want to read this item in the program we just say:

```
unsigned int NumberOfRuns

     = Configuration("Experiment", "NumberOfRuns");
```

Author: J.I. van Hemert
Version: 0.1

32.1 Retrieval operators

Overload operators

Names

virtual StringC const **operator)** (StringC const & iClass, StringC const & iItem) const
Configuration(string, string)

virtual StringC const **operator)** (char const * iClass, char const * iItem) const
Configuration(char, char*)*

virtual StringC const **operator)** (char const * iClass, StringC const & iItem) const
Configuration(char, string)*

virtual StringC const **operator)** (StringC const & iClass, char const * iItem) const
Configuration(string, char)*

Example:

```
StringC s;
s = Configuration("class", "variable");
StringC t("runs");
s = Configuration("experiment", t);
```

Return Value: String as requested type.
Author: J.I. van Hemert
Version: 0.1

33 class **ExperimentA**

Managing experiments

Public Members

33.1 **ExperimentA** (ConfigurationA const & iConfiguration, StatisticsA & iStatistics, CoreFactoryA const & iCoreFactory, ProblemFactoryA const & iProblemFactory, ReproductionFactoryA const & iReproductionFactory, RandomA & iRandom)
Declaring a new Experiment ... 105

33.2 virtual void **Run** () *Declaring a new Experiment* ... 106

Protected Members

static void **OutOfMemory** () *Handler for when the program runs out of memory. For now it only prints a message and exits.*

The experiment will runs the appropriate tests on different parameter settings for the problem and/or evolutionary algorithm. After running a test it saves the gathered information contained in then statistics class to a file.

Author: J.I. van Hemert
Version: 0.1

33.1

ExperimentA (ConfigurationA const & iConfiguration, StatisticsA & iStatistics, CoreFactoryA const & iCoreFactory, ProblemFactoryA const & iProblemFactory, ReproductionFactoryA const & iReproductionFactory, RandomA & iRandom)

Declaring a new Experiment

Example:

```
ExperimentA * Experiment = new ExperimentC(Configuration,
    Statistics, CoreFactory, ProblemFactory,
    ReproductionFactory, Random);
```

Return Value: Nothing.

Parameters:
`iConfiguration` — The configuration is used to determine settings for the algorithm and to provide a configuration to spawned objects within this class.
`iStatistics` — The statistics are used to write data to and build a report after a run of the evolutionary algorithm.
`iCoreFactory` — The factory that is used to spawn objects from classes of the Core package, within this class. It is passed on to these spawned objects as well.
`iProblemFactory` — The factory that is used to spawn objects from classes of the Problem package, within this class. It is passed on to these spawned objects as well.
`iReproductionFactory` — The factory that is used to spawn objects from classes of the Reproduction package, within this class. It is passed on to these spawned objects as well.
`iRandom` — This randomgenerator is given a seed and then passed on to underlying objects to serve as a source for random numbers.

Author: J.I. van Hemert

Version: 0.1

33.2

virtual void **Run** ()

Declaring a new Experiment

Example:

```
ExperimentA * Experiment = new ExperimentC(Configuration,
    Statistics, CoreFactory, ProblemFactory,
    ReproductionFactory, Random);

     ...

Experiment->Run();
```

This function should be implemented by the class inheriting from this one. It runs an experiment and collects statistical information. The experiment will consist of a problem and an evolutionary algorithm that tries to solve it. Here things as different parameter settings and multiple runs will be implemented.

Return Value:	Nothing.
Author:	J.I. van Hemert
Version:	0.1

34 class **StatisticsA**

Abstract statistics class

Public Members

Protected Members

double **TotalFitness** *Total of best fitnesses this run.*

unsigned long int **TotalSuccess** *Total number of successes this run.*

unsigned long int **NumberOfEvaluations** *Number of evaluations this run.*

unsigned long int **TotalNumberOfEvaluations** *Number of evaluations over all runs.*

Abstract base class defining the basic statistical data to be collected during experiments. These consist of *SuccessRate* (SR), *Average Evaluations to Solution* (AES) and *Mean Fitness* (MF).

Author: J.I. van Hemert
Version: 0.1

34.1

virtual void **ResetRunStatistics** ()

Reseting statistics

Example:

```
Statistics.ResetRunStatistics();
// Do some runs of the evolutionary algorithm on a problem.
cout << Statistics;
```

Call this to reset all the statistics that are used to collect data during runs.

Return Value: Nothing.
Author: J.I. van Hemert
Version: 0.1

34.2

virtual void **CollectBasicRunStatistics** (bool iIsSucces, FitnessType const & iBestFitness)

Collecting statistics

Example:

```
Statistics.ResetRunStatistics();
for (runs = 0; runs < 10; ++runs)
```

```
{
    // Run the evolutionary algorithm.
    Statistics.CollectBasicRunStatistics(SolutionFound,
       BestIndividual->GetFitness());
}
cout << Statistics;
```

After every run this function should be called to record the result of the last run.

Return Value:	Nothing.
Parameters:	`iSuccess` — Set this to `true` if the last run found a solution. `iBestFitness` — This should be the fitness of the best individual in population after the last run.
Author:	J.I. van Hemert
Version:	0.1

34.3 Registering Evaluations

Collecting statistics

Names

virtual void
RegisterEvaluation ()
Single evaluation.

virtual void
RegisterEvaluations (unsigned long int iEvaluations)
Multiple evaluations,

Example:

```
FitnessFunction::Calculate();
```

```
{
    // Calculate the fitness of some genome using its data.
    Statistics.RegisterEvaluation();
}
cout << Statistics;
```

These functions are used to register the amount of evaluations made by the fitnessfunction.

Return Value: Nothing.
Parameters: `iEvaluations` — Number of evaluations made, only needed when you want to register a number of evaluations in one single command.
See Also: FitnessFunctionA
Author: J.I. van Hemert
Version: 0.1

34.4

virtual unsigned long int **GetNumberOfEvaluations** () const

Collecting statistics

This function can be used in combination with the StopCondition. When a maximum number of evaluations is allowed, StopCondition can be used to terminate the current run when this number is reached.

Return Value: The number of evaluation done this run so far.
See Also: StopConditionA
Author: J.I. van Hemert
Version: 0.1

Class Graph