HeuristicLab

A Paradigm-Independent and Extensible Environment for Heuristic Optimization

Programming HeuristicLab

Algorithms and Problems

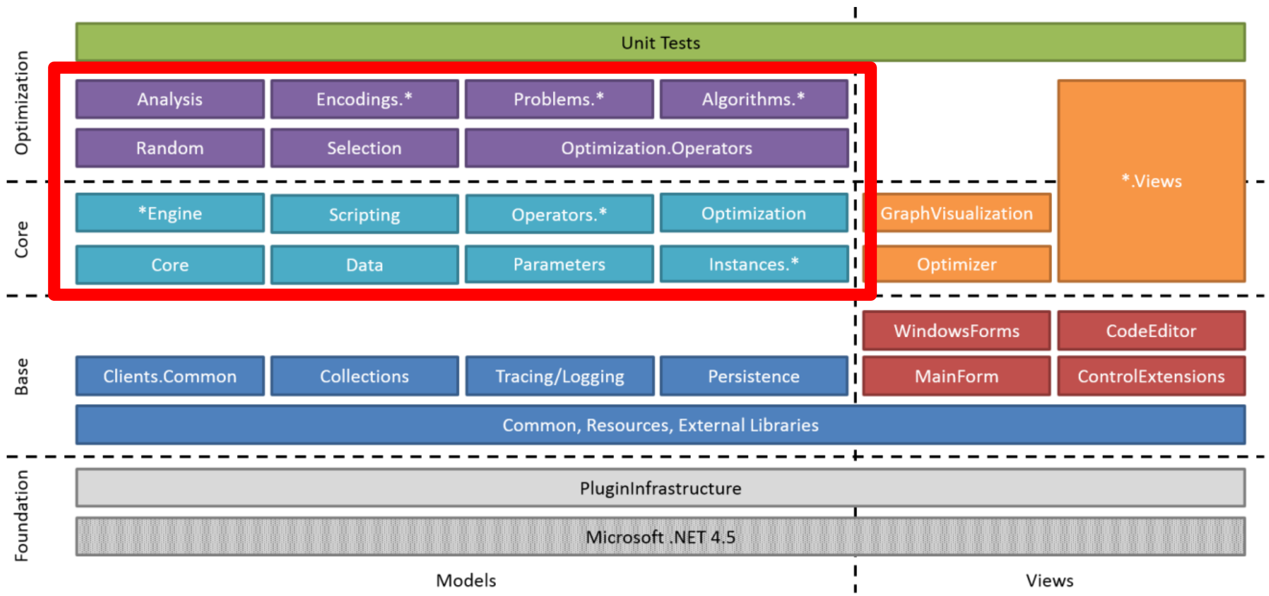
A. Scheibenpflug
Heuristic and Evolutionary Algorithms Laboratory (HEAL)
School of Informatics/Communications/Media, Campus Hagenberg
University of Applied Sciences Upper Austria

HEAL
Heuristic and Evolutionary Algorithms Laboratory

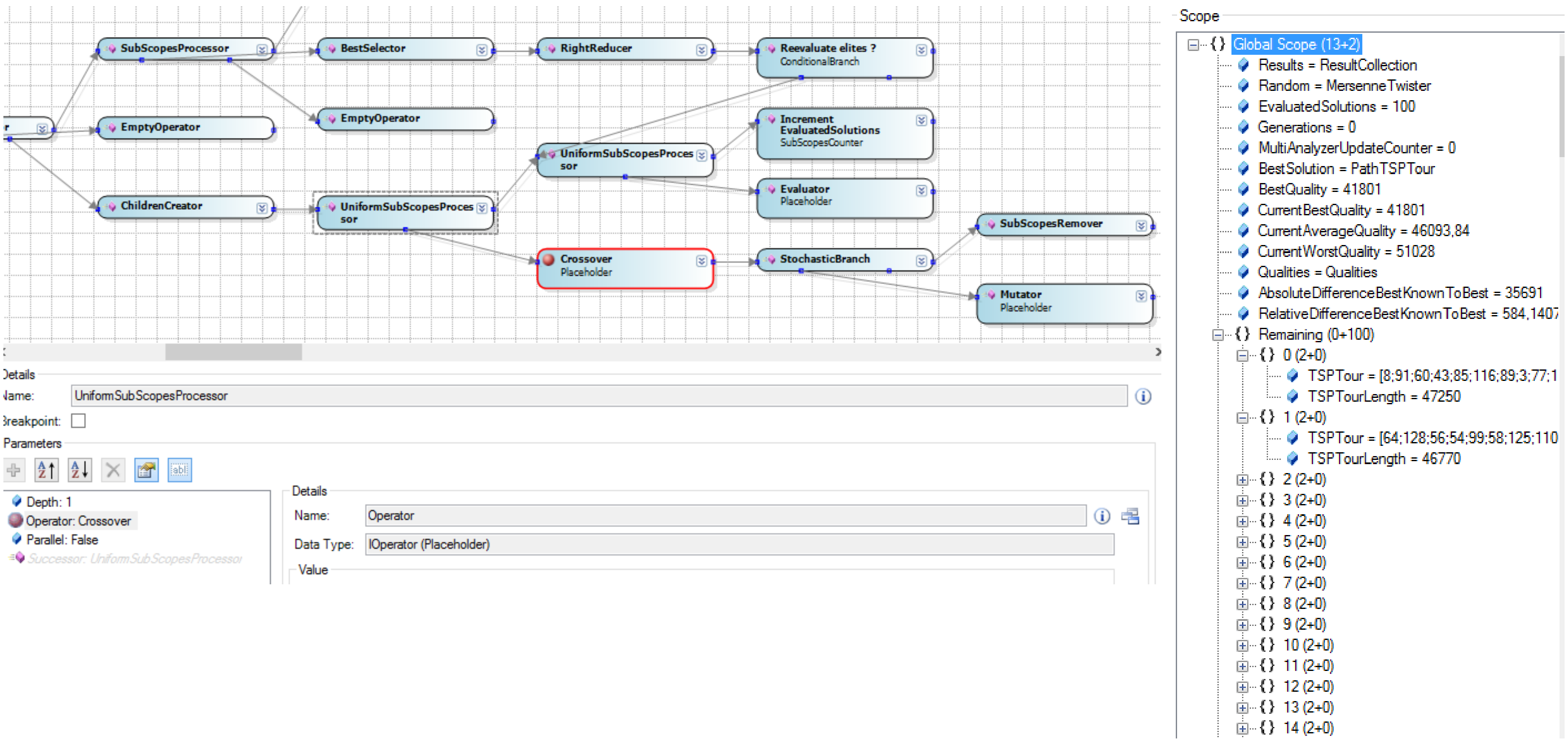
FH OBERÖSTERREICH

Overview

- HL Algorithm Model
- Parameters, Operators and Scopes
- Algorithms
- Problems



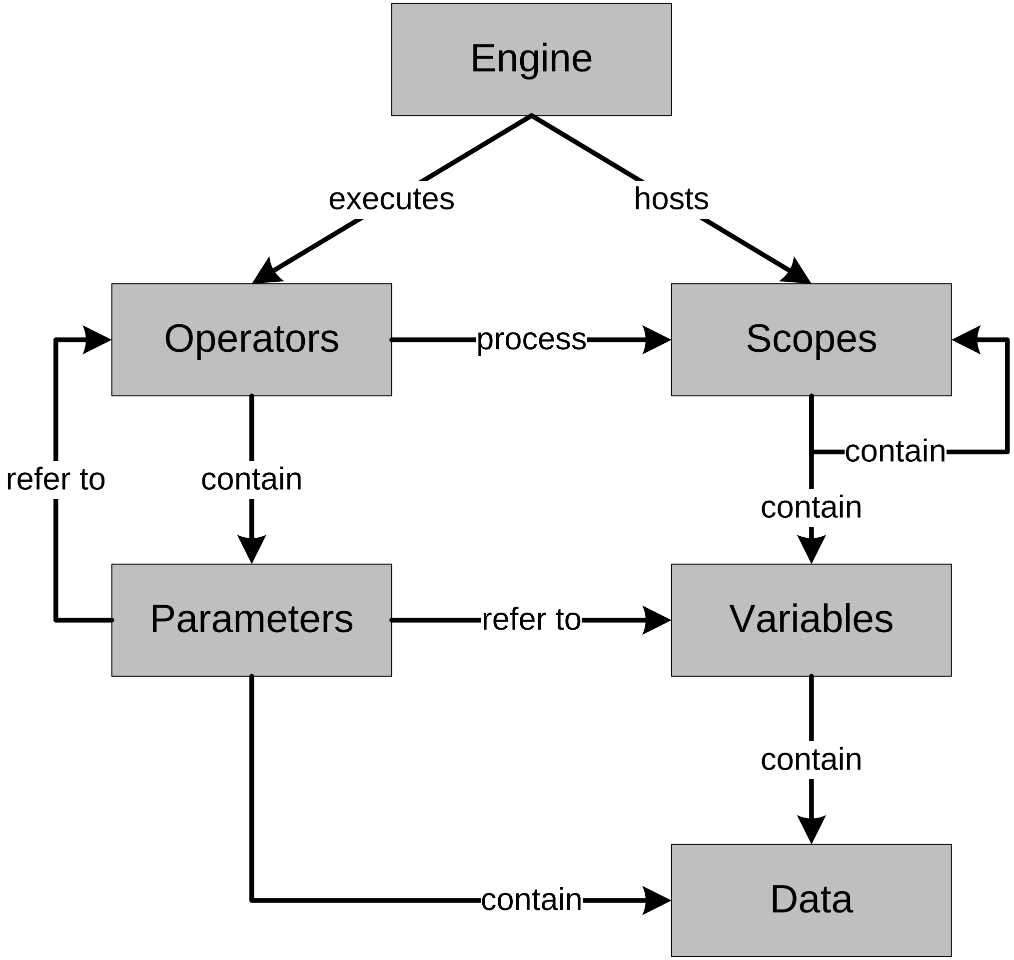
Parameters, Operators and Scopes



HL Algorithm Model

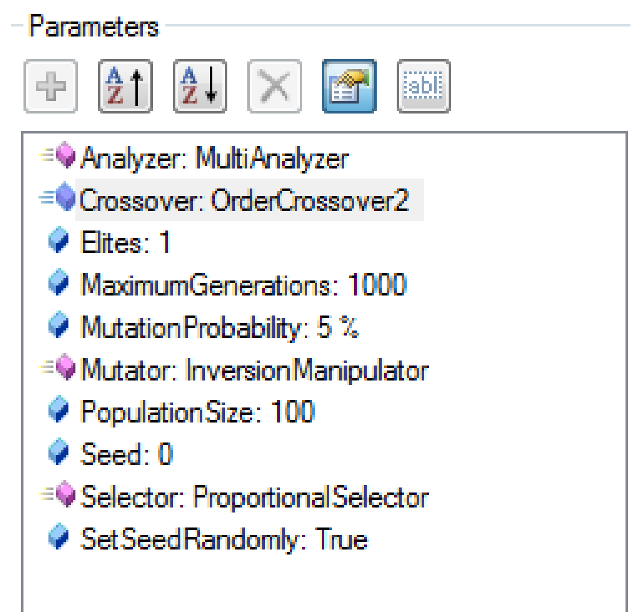
- Typically, HL algorithms are constructed by chaining together operators
- An engine executes these operators
  - Enables pausing and debugging
  - Available engines:
    - Sequential engine
    - Parallel engine
    - Debug engine
    - (Hive engine)

HL Algorithm Model



Parameters

- Used to configure algorithms, problems and operators
- Used for accessing variables in the scope
- E.g., population size, analyzers, crossover operator
- Operators
  - Look up these parameters from the algorithm, problem or scope
  - Use them to store values (in the scope tree)



Parameters

- `ValueParameter`:
  - Stores a value (Item) that can be looked up; e.g., mutation rate, crossover operator,…
- `LookupParameter`:
  - Looks up parameters/items (variables) from the scope/parent scopes.
- `ConstrainedValueParameter`:
  - Contains a list of selectable values.
- `ScopeTreeLookupParameter`:
  - Goes down the scope tree and looks up variables.
- `ScopeParameter`:
  - Returns the current scope.
- `ValueLookupParameter`, `OptionalConstrainedValueParameter`, `OperatorParameter`, `FixedValueParameter`, `OptionalValueParameter`,…

Parameters

- Everything that is a `ParameterizedNamedItem` has a parameters collection
- Normally used in the following way:
  - Add parameter to Parameters collection
  - Implement getter for convenience
  - Use parameter
  - Lookup parameter

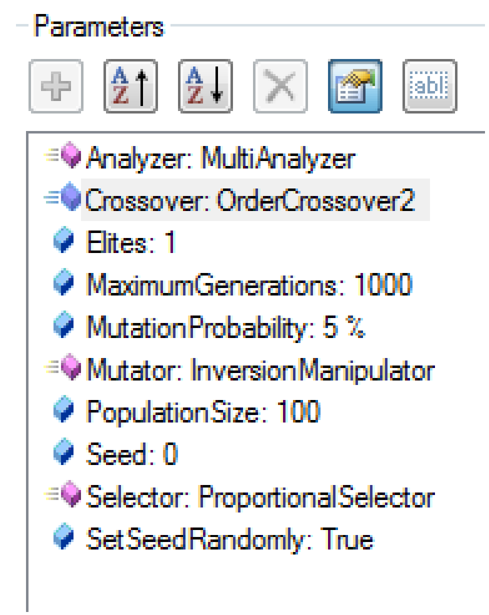
Add parameter to parameters collection

- The Crossover parameter enables the user to select different crossover operators:

```
Parameters.Add(new ConstrainedValueParameter<ICrossover>("Crossover",
"The operator used to cross solutions."));
```

- The PopulationSize is a freely configurable integer value:

```
Parameters.Add(new ValueParameter<IntValue>("PopulationSize",
"The size of the population of solutions.", new IntValue(100)));
```



Implement getter for convenience

- Getter for crossover parameter:

```
public IConstrainedValueParameter<ICrossover> CrossoverParameter {
          get { return (IConstrainedValueParameter<ICrossover>)Parameters["Crossover"]; }
}
```

- Getter for PopulationSize parameter:

```
private ValueParameter<IntValue> PopulationSizeParameter {
          get { return (ValueParameter<IntValue>)Parameters["PopulationSize"]; }
}
```

Use parameter

- Use crossover parameter:

```
ICrossover defaultCrossover = Problem.Operators.OfType<ICrossover>().FirstOrDefault();
foreach (ICrossover crossover in Problem.Operators.OfType<ICrossover>().OrderBy(x => x.Name))
          CrossoverParameter.ValidValues.Add(crossover);
CrossoverParameter.Value = defaultCrossover;
```

- Use PopulationSize parameter:

```
PopulationSizeParameter.Value.Value = 42;
```

Lookup Parameter

- Defining lookup parameter for crossover:

```
Parameters.Add(new ValueLookupParameter<IOperator>("Crossover",
"The operator used to cross solutions."));

public ValueLookupParameter<IntValue> PopulationSizeParameter {
          get { return (ValueLookupParameter<IntValue>)Parameters["PopulationSize"]; }
}
```

- Defining lookup parameter for population size:

```
Parameters.Add(new ValueLookupParameter<IntValue>("PopulationSize",
"The size of the population."));

public ValueLookupParameter<IOperator> CrossoverParameter {
          get { return (ValueLookupParameter<IOperator>)Parameters["Crossover"]; }
}
```

Use Lookup Parameter

- Set crossover parameter:

```
CrossoverParameter.Value =
ga.CrossoverParameter.ValidValues.Single(x => x.GetType() ==  typeof(OrderCrossover));
```

- Set PopulationSize parameter:

```
PopulationSizeParameter.Value.Value = 42;
```

Use Lookup Parameter

- In the genetic algorithm, a placeholder looks up the crossover that it executes:
  - Create placeholder

```
Placeholder crossover = new Placeholder();
```

  - Set the name of operator to lookup

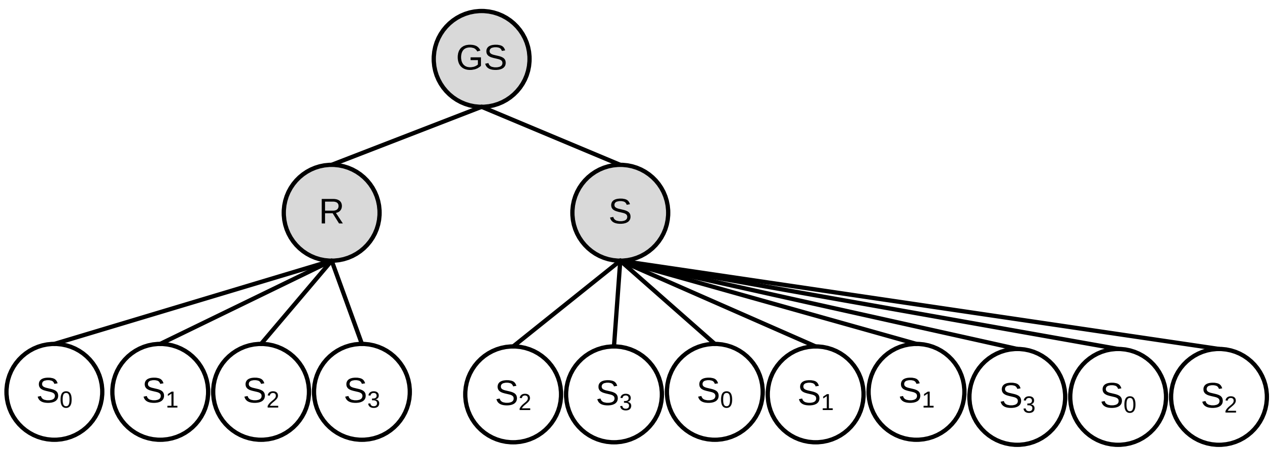
```
crossover.OperatorParameter.ActualName = "Crossover";
```

  - In the placeholder operator

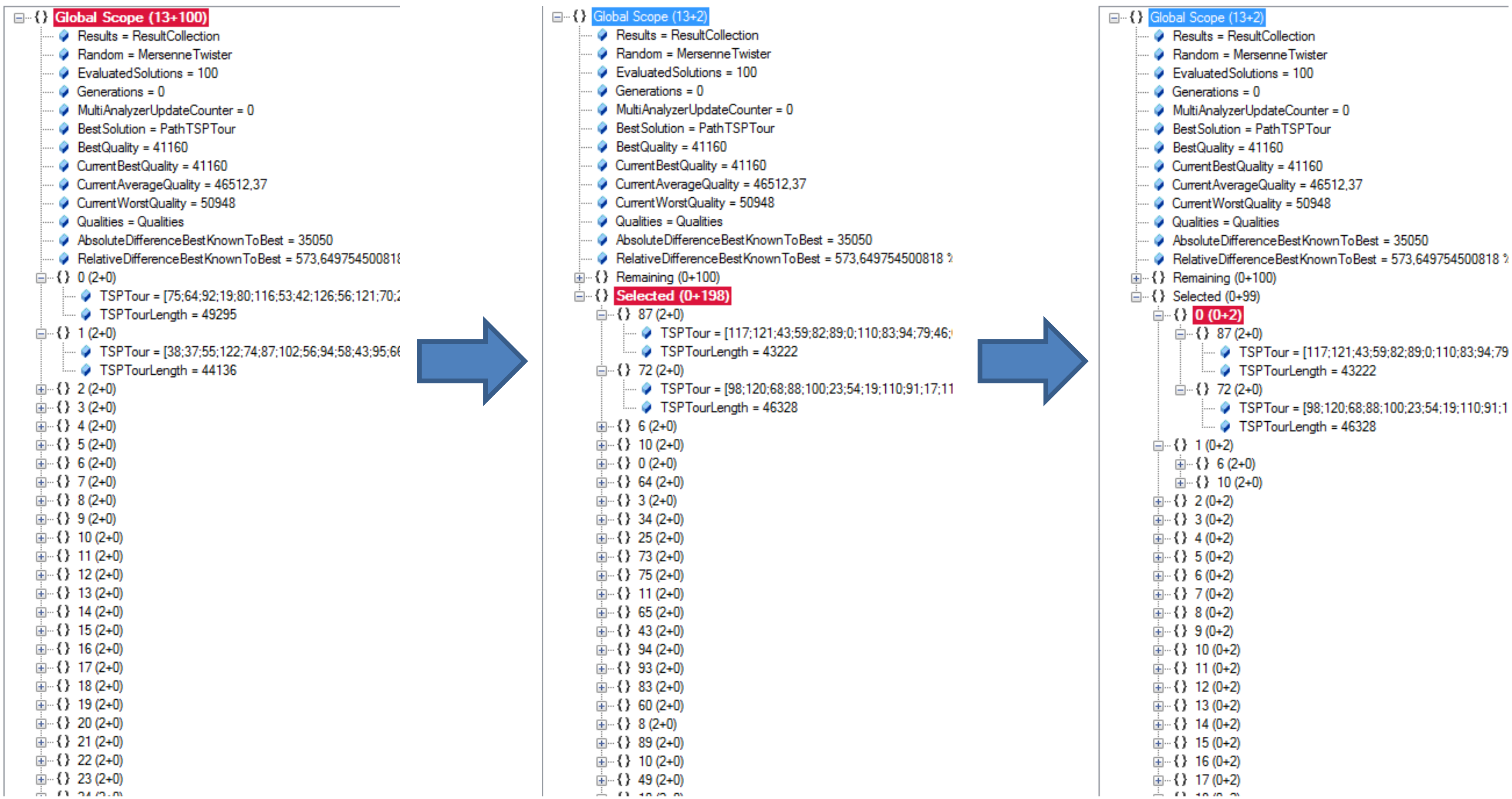
```
OperationCollection next = new OperationCollection(base.Apply());
IOperator op = OperatorParameter.ActualValue;
if (op != null)
    next.Insert(0, ExecutionContext.CreateOperation(op));
return next;
```

Scopes

- A scope is a node in the scope tree
- Contains link to parent and sub-scopes
- Contains variables (e.g., solutions or their quality)
- Operators usually work on scopes (either directly or through parameters)
- Example - Selection:



Scopes – Debug Engine

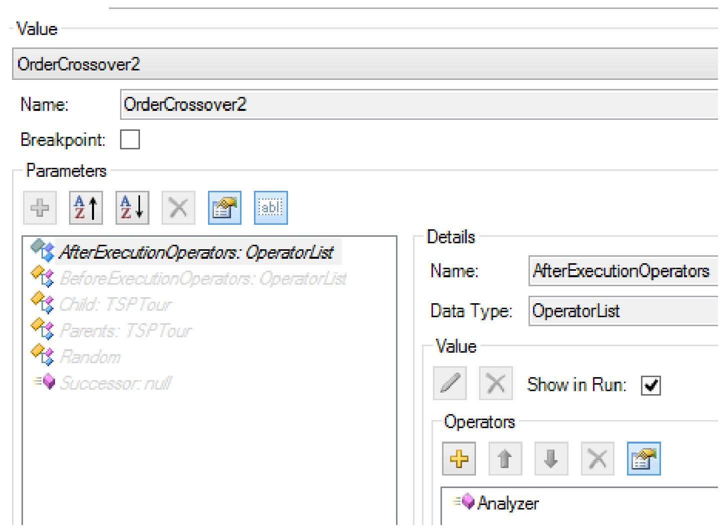


Operators

- Inherit from `SingleSuccessorOperator`
- Override the `Apply()` method
- Must return `base.Apply()`
  - Returns successor operation
- Use `ExecutionContext` to access scopes
- Or better: Use parameters to retrieve scopes, values from scopes or manipulate them

Instrumented Operators

- Inherit from `InstrumentedOperator`
- Override `InstrumentedApply()`
- Must return `base.InstrumentedApply()`
- Allows to configure before and after actions
- Useful for analyzers, additional functionality,… without changing the algorithm
- Think of aspect-oriented programming



Operators

A operator that increments a value from the scope by „Increment“

```
[Item("IntCounter", "An operator which increments an integer variable.")]
[StorableClass]
public sealed class IntCounter : SingleSuccessorOperator {
  public LookupParameter<IntValue> ValueParameter {
    get { return (LookupParameter<IntValue>)Parameters["Value"]; }
  }
  public ValueLookupParameter<IntValue> IncrementParameter {
    get { return (ValueLookupParameter<IntValue>)Parameters["Increment"]; }
  }
  public IntValue Increment {
    get { return IncrementParameter.Value; }
    set { IncrementParameter.Value = value; }
  }

  [StorableConstructor]
  private IntCounter(bool deserializing) : base(deserializing) { }
  private IntCounter(IntCounter original, Cloner cloner)
    : base(original, cloner) {
  }
  public IntCounter()
    : base() {
    Parameters.Add(new LookupParameter<IntValue>("Value", "The value which should be incremented."));
    Parameters.Add(new ValueLookupParameter<IntValue>("Increment", "The increment which is added to 
the value.", new IntValue(1)));
  }

  public override IDeepCloneable Clone(Cloner cloner) {
    return new IntCounter(this, cloner);
  }

  public override IOperation Apply() {
    if (ValueParameter.ActualValue == null) ValueParameter.ActualValue = new IntValue();
    ValueParameter.ActualValue.Value += IncrementParameter.ActualValue.Value;
    return base.Apply();
  }
}
```

For easier access to parameter values

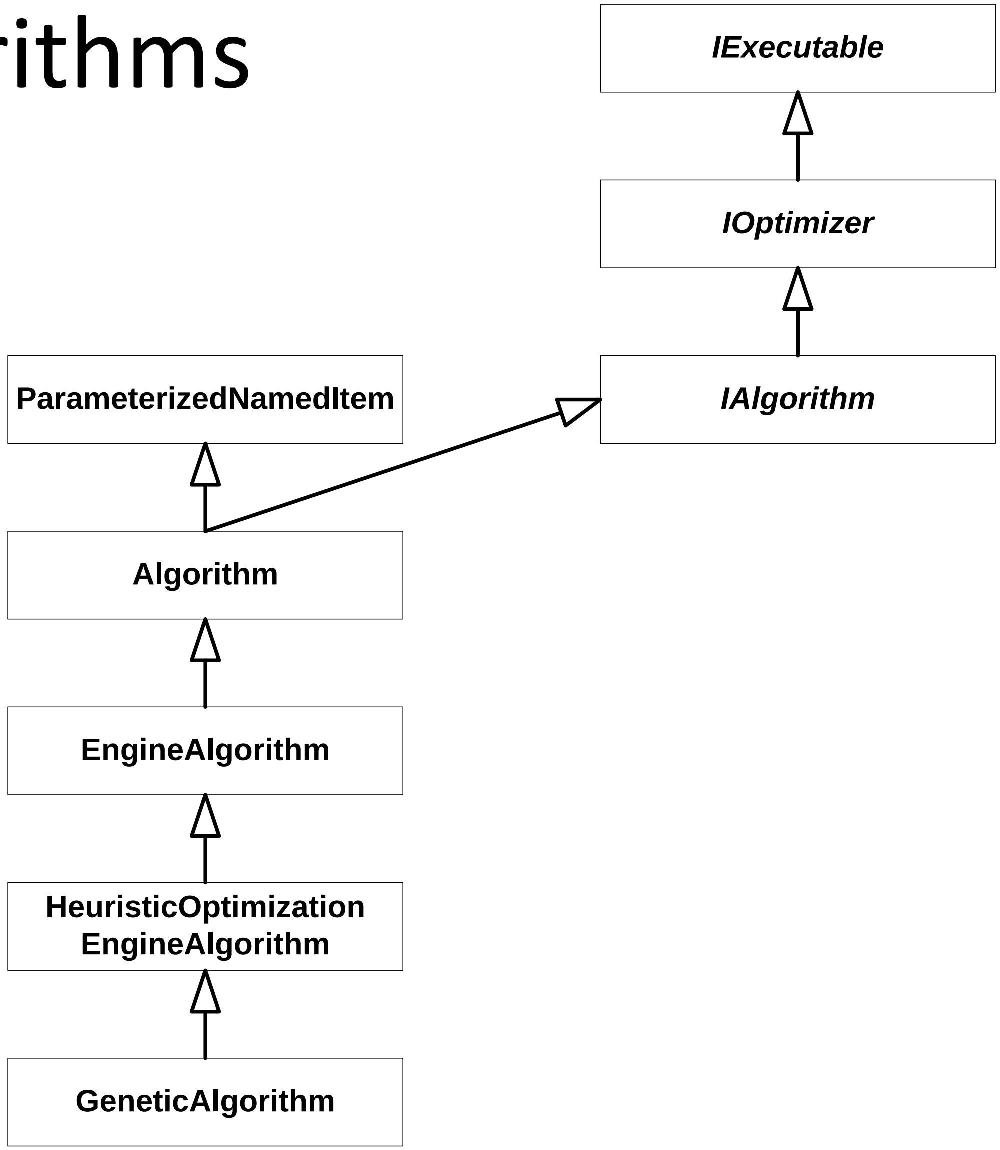
A parameter for retrieving „Value“ (default name, can be configure with ActualValue) from scope or parent scopes

If the value is not found it can also be created in the scope

Algorithms and Problems

- Different ways how to implement algorithms and problems
- Algorithms
  - Flexible: Inherit from `HeuristicOptimizationEngineAlgorithm`
  - Easy: Inherit from `BasicAlgorithm`
- Problems
  - Flexible: Inherit from `SingleObjectiveHeuristicOptimizationProblem`
  - Easy: Inherit from `[Single|Multi]ObjectiveBasicProblem`

Base classes/Interfaces for algorithms



Base classes/Interfaces for algorithms

- `IExecutable` (`Executable`):
  - Defines methods for starting, stopping, etc. of algorithms
- `IOptimizer`:
  - Contains a run collection
- `IAlgorithm`:
  - Contains a problem on which the algorithm is applied as well as a result
- `Algorithm`:
  - Base class, implements `IAlgorithm`
- `EngineAlgorithm`:
  - Extensions for execution with an engine (operator graph, scope, engine)
- `HeuristicOptimizationEngineAlgorithm`:
  - Specifies problem: `IHeuristicOptimizationProblem`

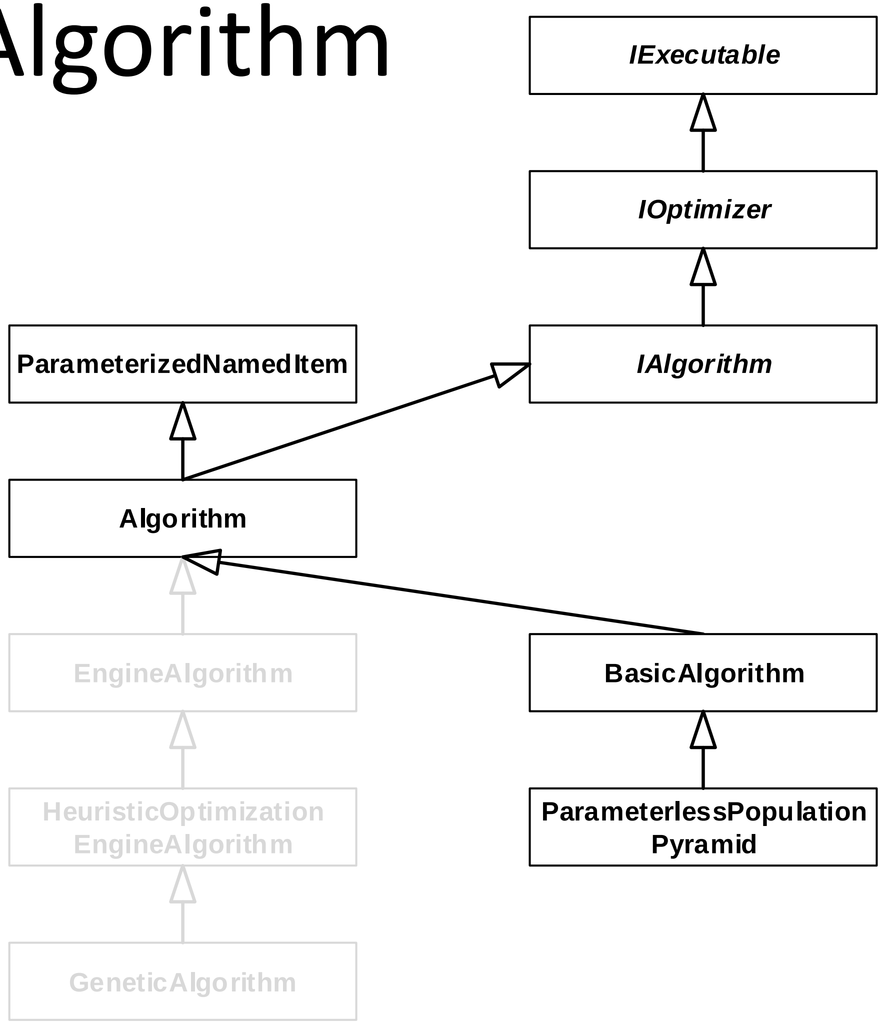
What does an HL algorithm do?

- Create operator graph of algorithm by chaining together operators (the actual algorithm)
- Offer user configuration options through parameters
- Discover operators from the operators collection of the problem/encoding
- Parameterize/wire (react to changes in operators) operators where necessary

BasicAlgorithm

- Creating an operator graph can be quite tricky
- Wiring operators is error-prone
- `BasicAlgorithms` are
  - Easy to implement
  - No boilerplate code
  - Hard-coded (no operator graph)
  - Don’t support pausing

Base classes/Interfaces for BasicAlgorithm



BasicAlgorithm - Interface

- Implement the Run method

```
protected override void Run(CancellationToken cancellationToken)
```

- Optional: Fix problem type

```
public override Type ProblemType {
   get { return typeof(BinaryProblem); }
}

public new BinaryProblem Problem {
   get { return (BinaryProblem)base.Problem; }
   set { base.Problem = value; }
}
```

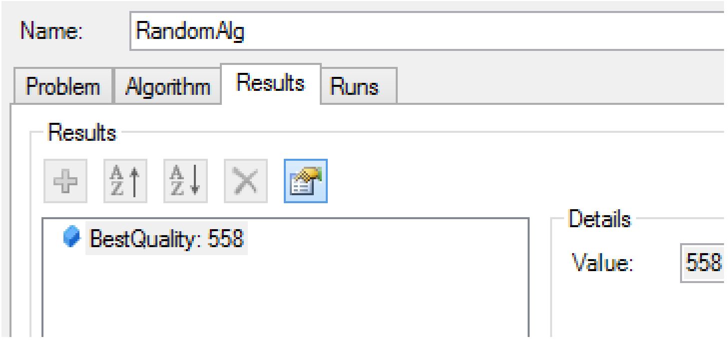
Example – Random Search

```
protected override void Run(CancellationToken cancellationToken)  {
      DoubleValue bestQuality = new DoubleValue(0.0);
      Results.Add(new Result("BestQuality", bestQuality));

      for(int i = 0; i < 100000; i++) {
         cancellationToken.ThrowIfCancellationRequested();

         BinaryVector b = new BinaryVector(Problem.Length, random);
         double curQuality = Problem.Evaluate(b, random);

         if(Problem.Maximization && curQuality > bestQuality.Value) {
           bestQuality.Value = curQuality;
         } else if(!Problem.Maximization && curQuality < bestQuality.Value) {
           bestQuality.Value = curQuality;
         }
      }
}
```



Problems

- Use encodings for representing solutions
- Encodings consist of solution candidate definitions and corresponding operators
- Problems contain
  - the evaluator
  - the solution creator
- Define maximization or minimization
- Contain the „problem data“ (e.g., a distance matrix, a simulation, a function definition), usually supplied by a problem instance provider
- Can be single- or multi-objective
- Configured with parameters

Problem Architecture

Problem

e.g. Vehicle Routing, Quadratic Assignment, Symbolic Regression,...

Operators

Evaluators, Move Evaluators, Creators, Crossover, Manipulators, Move Generators, Move Makers, Particle Operators

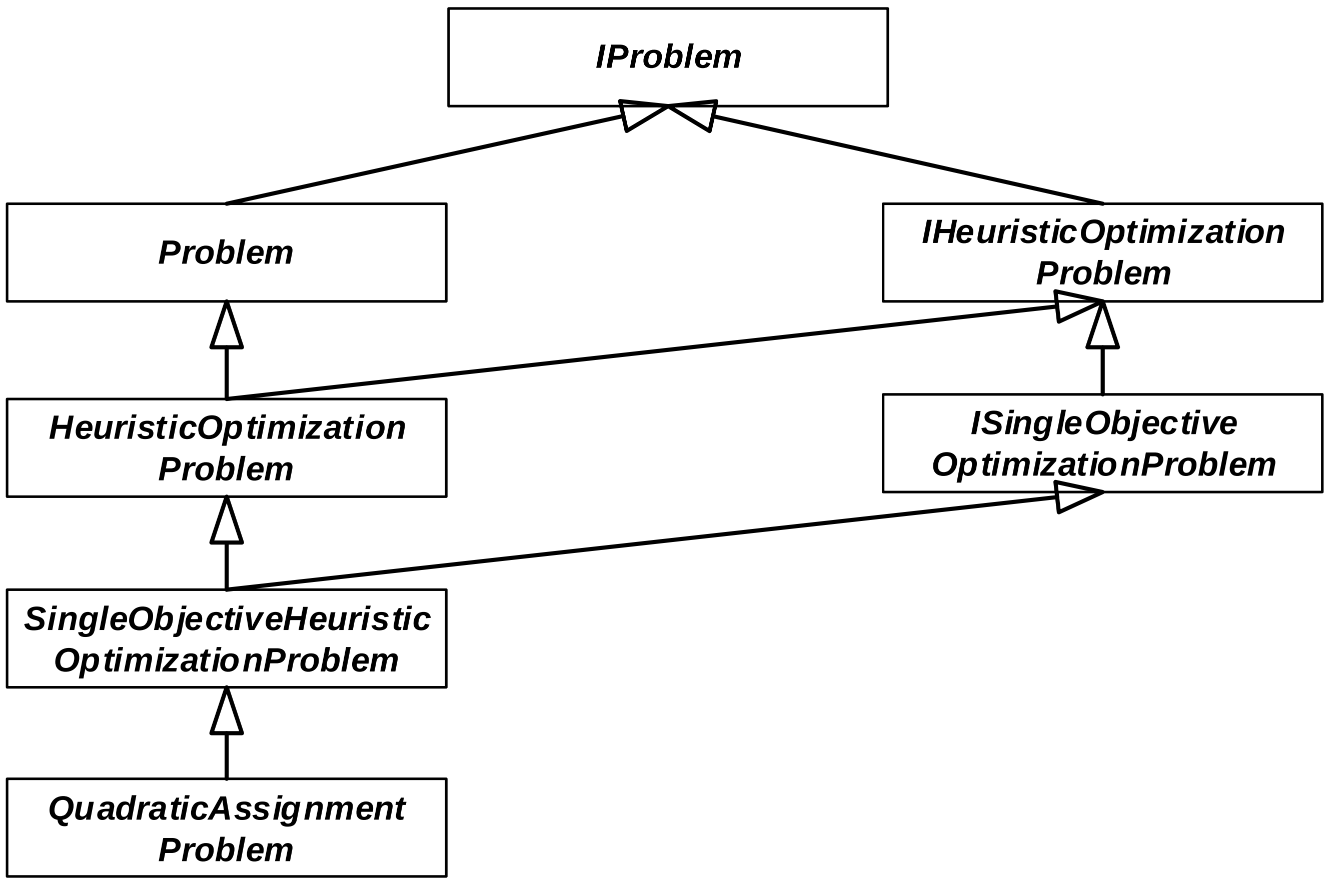
Encoding

e.g. Permutation, RealVector, Binary,...

Operators

Creators, Crossover, Manipulators, Move Generators, Move Makers, Particle Operators

Base classes/Interfaces for problems



Base classes/Interfaces for problems

- `IProblem`:
  - Contains the operators collection; all operators that can be used by the problem, algorithm and user
- `IHeuristicOptimizationProblem`:
  - Defines solution creator and evaluator
- `Problem`, `HeuristicOptimizationProblem` and `Single/MultiObjectiveHeuristicOptimizationProblem` provide abstract base classes

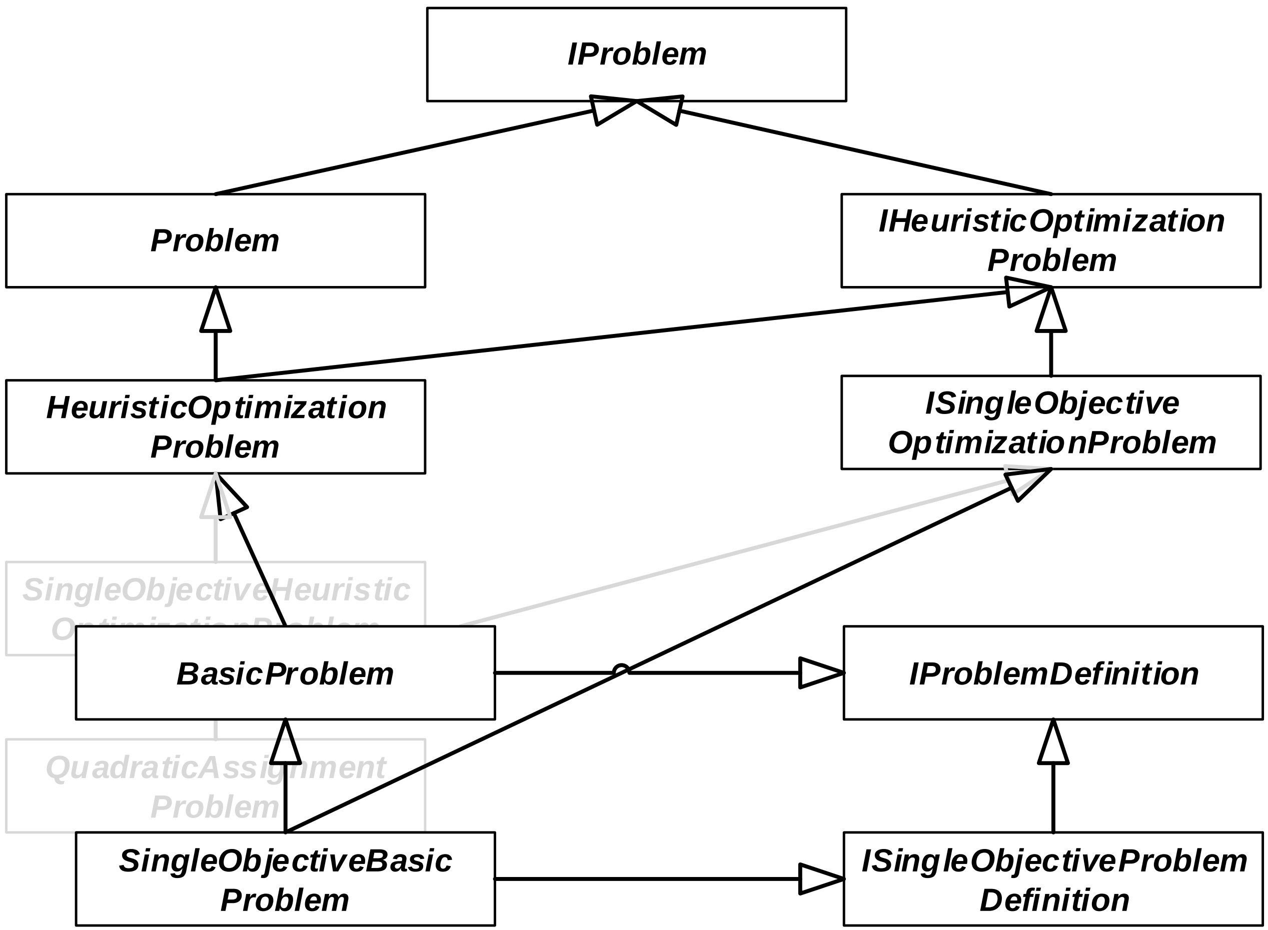
Recap: What does a HL problem do?

- Defines used encoding
- Defines single/multi objective
- Defines min/maximization
- Discovers correct operators
  - Are used by the algorithm
- Wires/parameterizes operators
- Loads problem data using a corresponding problem instance provider

BasicProblem

- Similar concept as `BasicAlgorithm`
- Makes implementing new problems easier
- No wireing/operators necessary
- Use automatic encoding configuration
- Don’t work with all algorithm types, e.g., algorithms that use very specific operators
  - Simulated Annealing
  - Scatter Search
  - Particle Swarm Optimization

Base classes/Interfaces for BasicProblem



BasicProblem - Interface

- Define encoding

```
MyNewProblem : SingleObjectiveBasicProblem<BinaryVectorEncoding>
```

- Define maximization or minimization

```
bool Maximization { get; }
```

- Evaluate a solution and return quality

```
double Evaluate(Individual individual, IRandom random);
```

BasicProblem - Interface

- Until now only GA variants can use the problem
- Implement neighbourhood function to also use trajectory-based metaheuristics

```
IEnumerable<Individual> GetNeighbors(Individual individual, IRandom random);
```

- Optional: Add analysis code for tracking results

```
void Analyze(Individual[] individuals, double[] qualities, ResultCollection results,
IRandom random);
```

BasicProblem – Example: OneMax

```
class OneMaxProblem : SingleObjectiveBasicProblem<BinaryVectorEncoding> {
      public OneMaxProblem() { }
      [StorableConstructor]
      protected OneMaxProblem(bool deserializing) : base(deserializing) {
}
      public OneMaxProblem(OneMaxProblem alg, Cloner cloner) : base(alg,
cloner) { }
      public override IDeepCloneable Clone(Cloner cloner) {
        return new OneMaxProblem(this, cloner);
      }

      public override bool Maximization { get{ return true; } }

      public override double Evaluate(Individual individual,
                                      IRandom random) {
        return individual.BinaryVector().Count(b => b);
      }
   }
```

Useful Links

http://dev.heuristiclab.com/trac.fcgi/wiki/Documentation

http://dev.heuristiclab.com/trac.fcgi/wiki/Research

heuristiclab@googlegroups.com

http://www.youtube.com/heuristiclab