

HeuristicLab 3.0

Overview

Motivation

- genericity
 - no focus on any particular heuristic optimization paradigm
 - suitable for any kind of algorithm
 - suitable for any kind of problem and solution representation
- modularity
 - plug-ins
- usability
 - extensive use of graphical user interfaces (WinForms)
 - step-wise algorithm execution
 - persistence
 - interactive algorithm engineering
- parallelism
 - integration of parallel algorithms
 - separation of algorithms and parallelization concept

Plugin Infrastructure

```
using HeuristicLab.PluginInfrastructure;

namespace MyProject {
    [ClassInfo(Name = "My Project", Version = "0.99.0.0", Description = "Just an example plugin")]
    [PluginFile(Filename = "MyProject.dll", Filetype = PluginFileType.Assembly)]
    [PluginFile(Filename = "ZipLibrary.dll", Filetype = PluginFileType.Assembly)]
    [PluginFile(Filename = "Database.db", Filetype = PluginFileType.Data)]

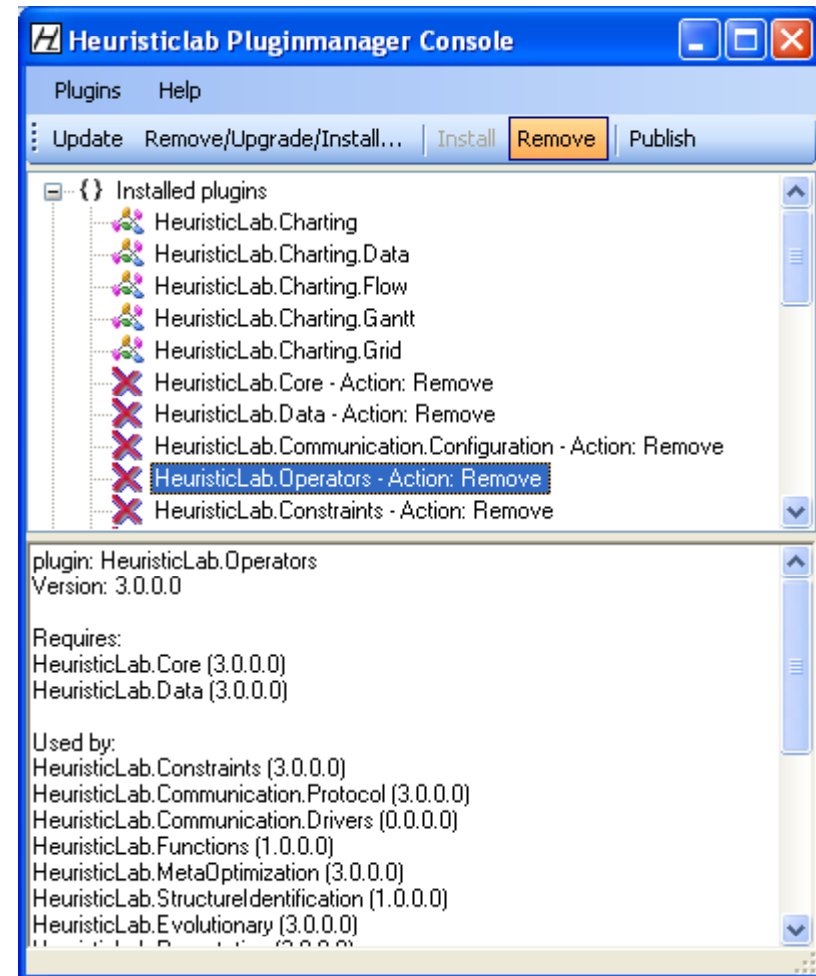
    [Dependency(Dependency = "HeuristicLab.Data")]
    [Dependency(Dependency = "HeuristicLab.Random")]

    public class MyProjectPlugin : PluginBase {
    }
}
```

- a plugin can contain multiple files (assemblies, data files, images, icons, ...)
- plugins may depend on each other
- plugin meta-data (descriptions, dependencies, etc.) are stored in the code files (attributes)
- no XML description file like in HeuristicLab 2.0
- plugin infrastructure enables drop-in installation (similar to eclipse)

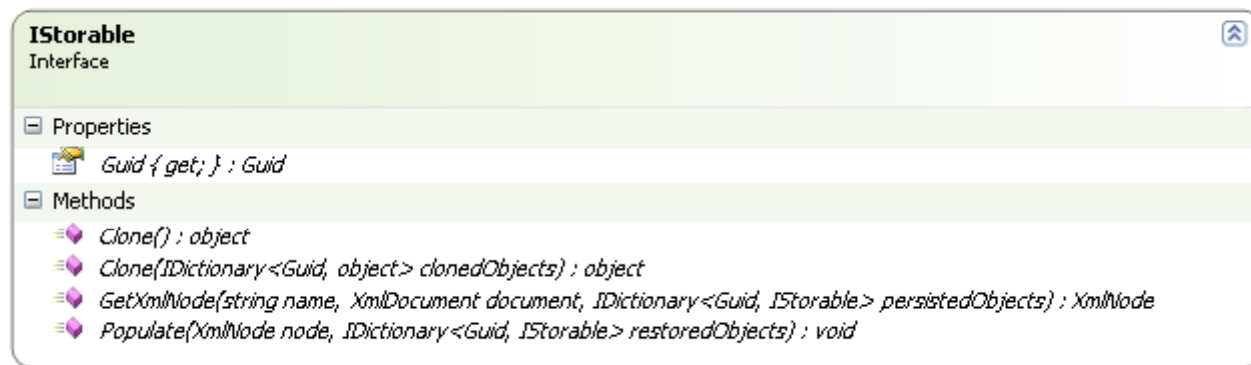
Plugin Infrastructure

- plugin manager
 - inspired by Debian package management system
 - plugins are deployed as ZIP archives
 - plugin manager minds dependencies
 - required plugins are automatically installed
 - dependent plugins are automatically removed
 - HTTP-based updating of plugins
 - plugins can be installed and removed at runtime
 - each application has its own application domain



Persistence

- basic interface **IStorable**
 - defines unique identifier (GUID)
 - Guid **Guid** { get; }
 - defines methods for cloning
 - object **Clone**()
 - object **Clone**(IDictionary<Guid, object> clonedObjects)
 - defines methods for persisting to and restoring from XML
 - XmlNode **GetXmlNode**(string name, XmlDocument document, IDictionary<Guid, IStorable> persistedObjects)
 - void **Populate**(XmlNode node, IDictionary<Guid, IStorable> restoredObjects)
 - implemented by abstract base class **StorableBase**
- multiple references to a single object are handled correctly
- cyclical object graphs can be cloned or persisted automatically
- deep cloning is used by default



The screenshot shows the definition of the **IStorable** interface. It is categorized as an "Interface" and is organized into two sections: "Properties" and "Methods".

Properties:

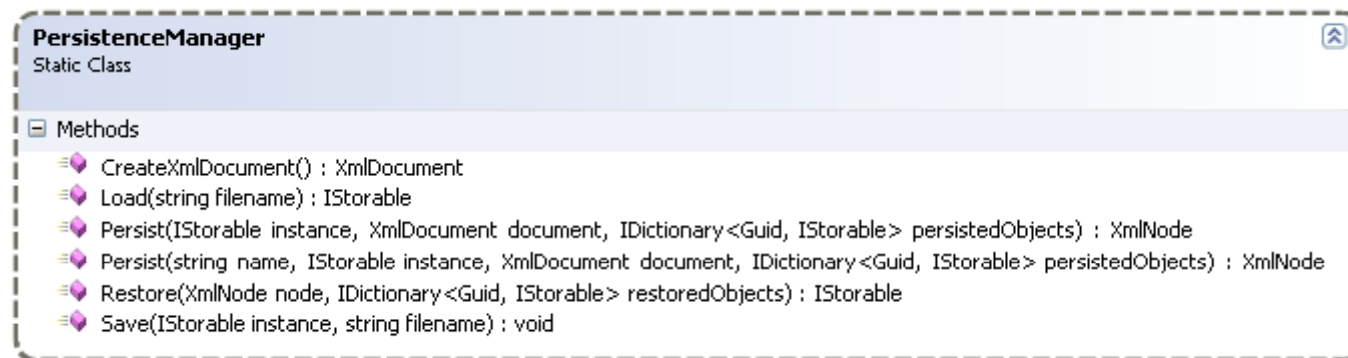
- `Guid { get; } : Guid`

Methods:

- `Clone() : object`
- `Clone(IDictionary<Guid, object> clonedObjects) : object`
- `GetXmlNode(string name, XmlDocument document, IDictionary<Guid, IStorable> persistedObjects) : XmlNode`
- `Populate(XmlNode node, IDictionary<Guid, IStorable> restoredObjects) : void`

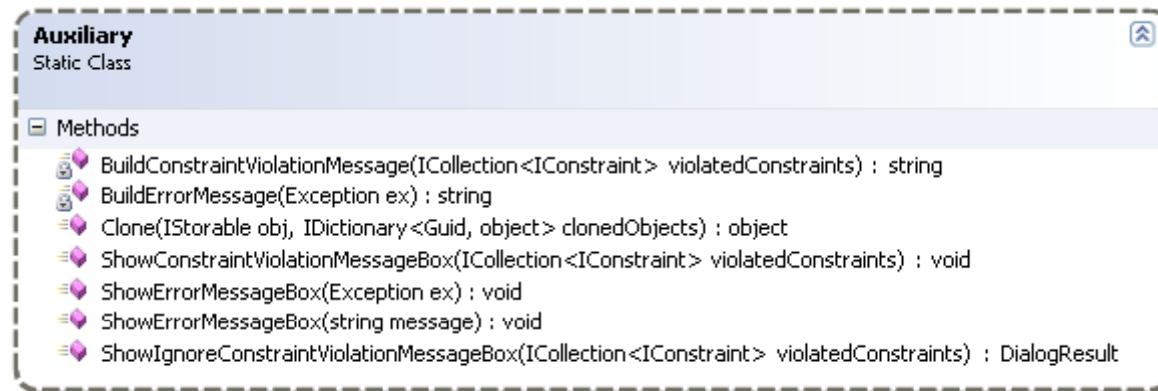
Persistence

- static class **PersistenceManager**
 - provides auxiliary methods for persistence
 - **Load** and **Save** to write IStorable to or to restore it from disk
 - **Persist** and **Restore** to start persisting or restoring of single IStorable
 - by default type names are used as XML tags
 - Persist offers parameter (name) to specify some other name as XML tag



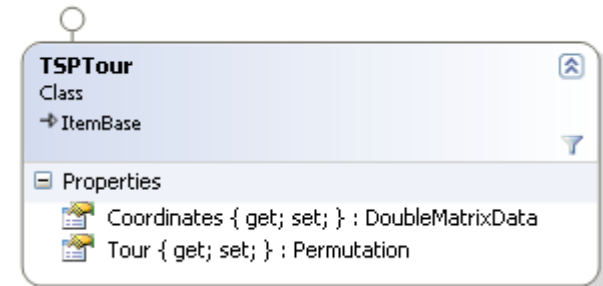
Persistence

- static class **Auxiliary**
 - provides auxiliary method for cloning
 - **Clone** to start cloning of a single IStorable
 - checks if cloning is necessary or if object has already been cloned
 - provides several other methods for showing error messages
- methods of PersistenceManager and Auxiliary have to be used to enable correct persisting / cloning of cyclical object graphs



Persistence

- persistence code in class TSPTour



```
public override XmlNode GetXmlNode(string name, XmlDocument document, IDictionary<Guid, IStorable> persistedObjects) {
    XmlNode node = base.GetXmlNode(name, document, persistedObjects);
    node.AppendChild(PersistenceManager.Persist("Coordinates", Coordinates, document, persistedObjects));
    node.AppendChild(PersistenceManager.Persist("Tour", Tour, document, persistedObjects));
    return node;
}
```

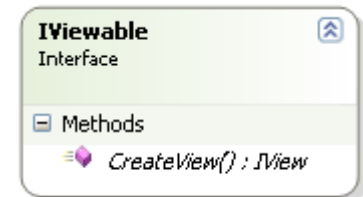
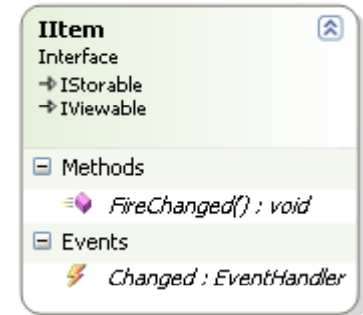
```
public override void Populate(XmlNode node, IDictionary<Guid, IStorable> restoredObjects) {
    base.Populate(node, restoredObjects);
    myCoordinates = (DoubleMatrixData)PersistenceManager.Restore(node.SelectSingleNode("Coordinates"), restoredObjects);
    myTour = (Permutation)PersistenceManager.Restore(node.SelectSingleNode("Tour"), restoredObjects);
}
```

- example XML snippet

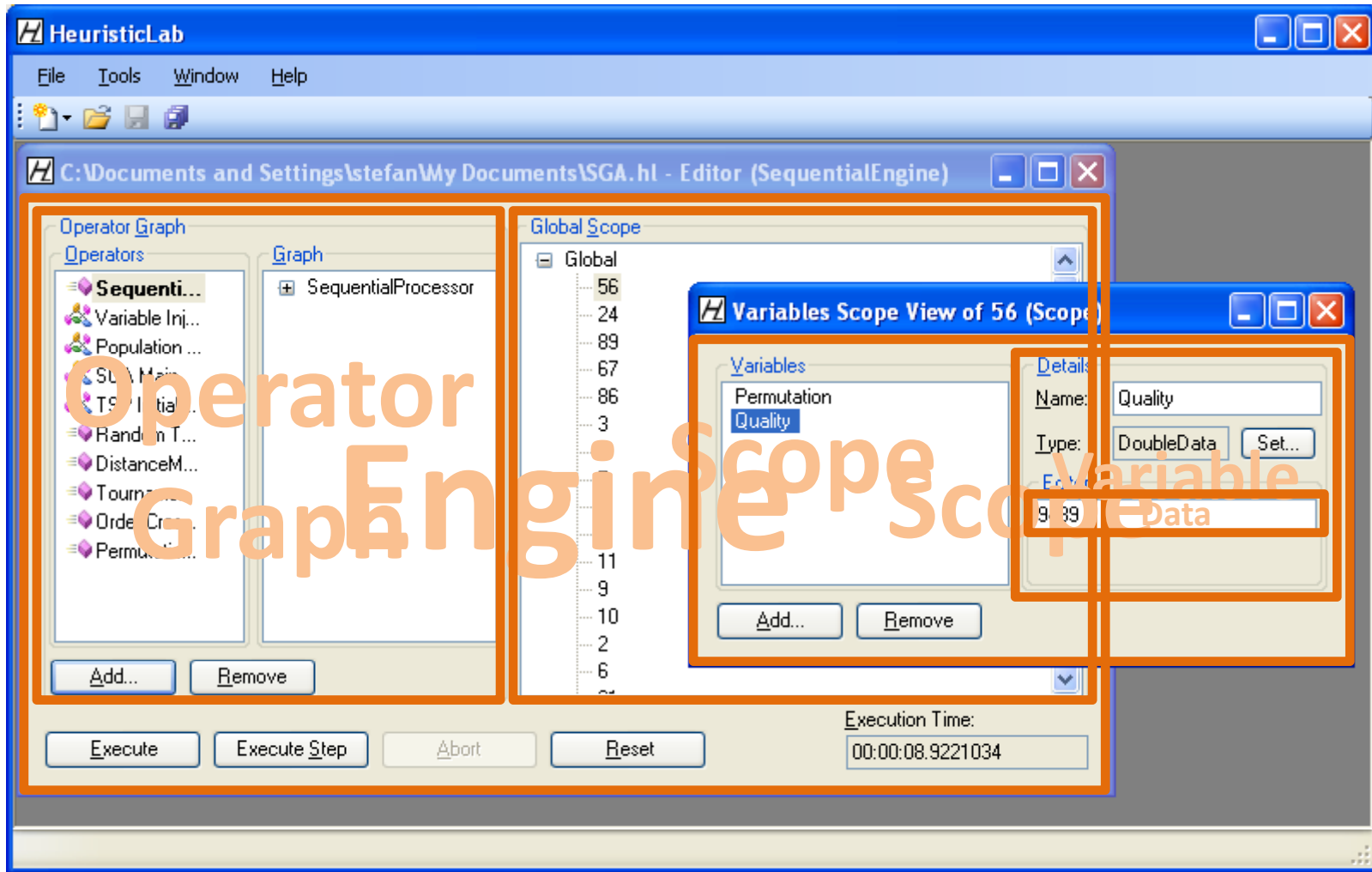
```
<VariableInjector Type="HeuristicLab.Operators.VariableInjector, HeuristicLab.Operators" GUID="65e6153d-18a6-4a9a-9b04-77732e8612b6"
Name="VariableInjector">
  <SubOperators />
  <VariableInfos />
  <Variables>
    <Variable Type="HeuristicLab.Core.Variable, HeuristicLab.Core" GUID="d971503d-2ed0-43b2-bc6b-1bce4676dc51" Name="PopulationSize">
      <Value Type="HeuristicLab.Data.IntData, HeuristicLab.Data" GUID="0f36459e-1910-4852-91a6-ec37a9767df7">100</Value>
    </Variable>
    <Variable Type="HeuristicLab.Core.Variable, HeuristicLab.Core" GUID="cb899fcb-76c3-4f86-9699-91cb6ee62420" Name="EvaluatedSolutions">
      <Value Type="HeuristicLab.Data.IntData, HeuristicLab.Data" GUID="4bd5b9ae-4ee2-4063-8e17-37972eae50b7">0</Value>
    </Variable>
  </Variables>
</VariableInjector>
```

Items

- interface **IItem**
 - implements IStorable and IViewable
 - base class for (almost) every HeuristicLab object
 - data, variables, operators, engines, ...
 - defines **Changed** event
 - defines method **FireChanged** to fire Changed event manually
 - has to be called if contained complex objects are changed and don't provide an event-based notification (arrays, e.g.)
 - implemented by abstract base class **ItemBase**
- interface **IViewable**
 - defines method **CreateView** for creating a view for an object
 - usually a view is a Windows Forms control
 - implemented by base class **ViewBase**
- all objects implementing IItem can be shown on the user interface (GUI)
- views of complex objects can be assembled using views of their parts
- strict use of MVC pattern



Views



Modeling Algorithms

- what is an algorithm?

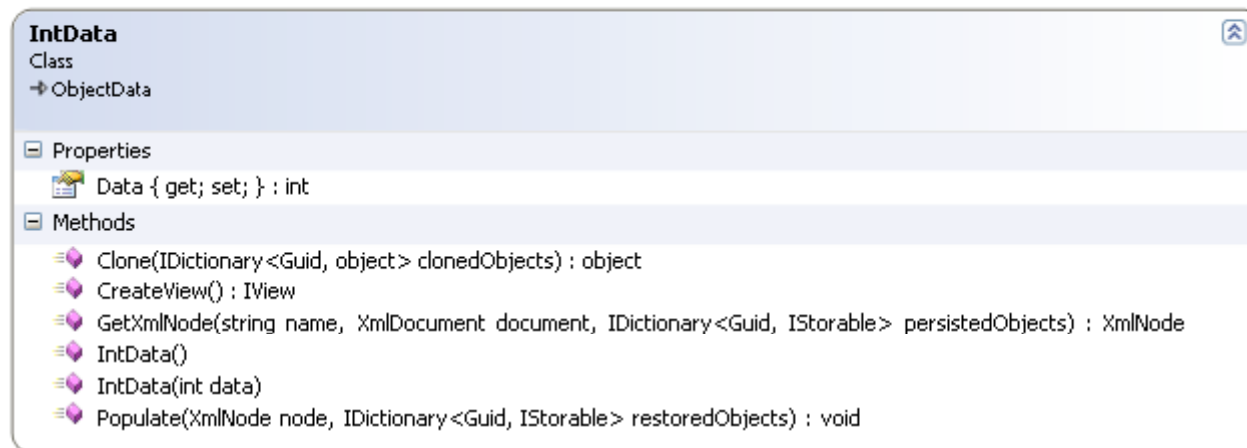
"A finite set of unambiguous instructions performed in a prescribed sequence to achieve a goal, especially a mathematical rule or procedure used to compute a desired result. Algorithms are the basis for most computer programming."

© The American Heritage Dictionary of the English Language

- 3 main aspects
 - instructions
 - sequence of execution
 - result
- we have to model
 - operators modifying data
 - graphs of operators representing the sequence of execution

Data

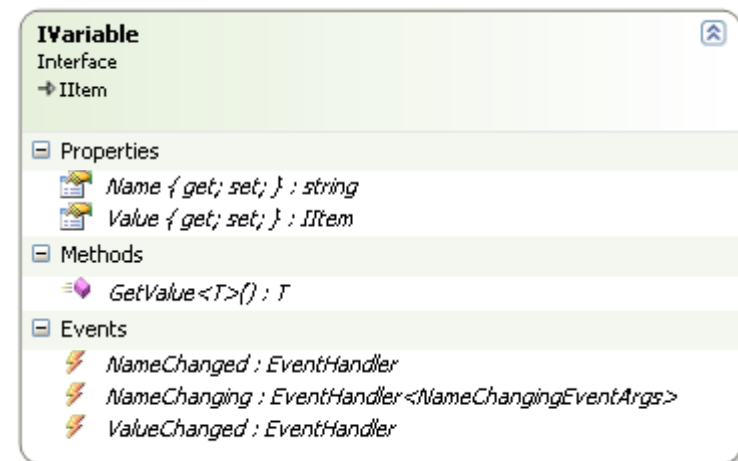
- data is represented as objects
 - data objects have to implement `IItem` (are storable and viewable)
 - persistence methods have to be implemented
 - custom views can be provided
- namespace **HeuristicLab.Data** contains several ready to use data objects
 - wrapper objects for system types
 - `IntData`, `IntArrayData`, `IntMatrixData`, `DoubleData`, ...
 - collections
 - `ItemList`



The screenshot shows the `IntData` class in Visual Studio. The class is a `Class` that inherits from `ObjectData`. It has a `Data` property of type `int` with `get` and `set` methods. The `Methods` section lists several methods: `Clone(IDictionary<Guid, object> clonedObjects) : object`, `CreateView() : IView`, `GetXmlNode(string name, XmlDocument document, IDictionary<Guid, IStorable> persistedObjects) : XmlNode`, `IntData()`, `IntData(int data)`, and `Populate(XmlNode node, IDictionary<Guid, IStorable> restoredObjects) : void`.

Variables

- operators manipulate data
- data objects need names to be accessible
- interface **IVariable**
 - contains name (**Name**) and data object (**Value**)
 - defines generic method **GetValue** to access value
 - various events to signal changes (MVC)
 - implemented by **Variable**
- IVariable implements IItem
 - variables themselves can be values of variables
 - required for meta-programming
 - more on that later ...



The screenshot displays the definition of the **IVariable** interface in an IDE. The interface is shown as a light green box with a title bar. It includes the following details:

- IVariable**
Interface
→ IItem
- Properties**
 - Name* { get; set; } : string
 - Value* { get; set; } : IItem
- Methods**
 - GetValue*<T>() : T
- Events**
 - NameChanged* : EventHandler
 - NameChanging* : EventHandler<NameChangingEventArgs>
 - ValueChanged* : EventHandler

Scopes

- variables are organized in scopes
- interface **IScope**
 - hierarchical container of variables
 - contains variables (**Variables**)
 - contains sub-scopes (**SubScopes**)
 - defines several methods to get/add/remove variables and sub-scopes
 - defines method **GetVariableValue** to perform recursive variable lookup towards the root scope
 - variables with identical names hide variables of parent scopes
 - events to signal changes (MVC)
 - implemented by **Scope**
- scopes build an n-ary tree structure
- operators work on scopes
 - get variables, manipulate, write back

The screenshot displays the **IScope** interface in an IDE. The interface is defined as follows:

```
IScope
Interface
↳ IItem

Properties
  Name { get; } : string
  SubScopes { get; } : IList<IScope>
  Variables { get; } : ICollection<IVariable>

Methods
  AddSubScope(IScope scope) : void
  AddVariable(IVariable variable) : void
  Clear() : void
  GetScope(Guid guid) : IScope
  GetScope(string name) : IScope
  GetVariable(string name) : IVariable
  GetVariableValue(string name, bool recursiveLookup) : IItem
  GetVariableValue(string name, bool recursiveLookup, bool throwOnError) : IItem
  GetVariableValue<T>(string name, bool recursiveLookup) : T
  GetVariableValue<T>(string name, bool recursiveLookup, bool throwOnError) : T
  RemoveSubScope(IScope scope) : void
  RemoveVariable(string name) : void
  ReorderSubScopes(int[] sequence) : void
  SetParent(IScope scope) : void

Events
  SubScopeAdded : EventHandler<ScopeIndexEventArgs>
  SubScopeRemoved : EventHandler<ScopeIndexEventArgs>
  SubScopesReordered : EventHandler
  VariableAdded : EventHandler<VariableEventArgs>
  VariableRemoved : EventHandler<VariableEventArgs>
```

Operators

- operators represent basic instructions of algorithms
- operators manipulate scopes (modify variables and/or sub-scopes)
- operators decide which operators are executed next
- interface **IOperator**
 - contains sub-operators (**SubOperators**)
 - contains local variables (**Variables**)
 - contains meta-information about variables (**VariableInfos**)
 - defines methods to manipulate its data
 - events to signal changes
 - defines method **Execute** to execute the operator on a specific scope
 - defines **GetVariableValue** methods for convenient variable lookup (either in the local variables or in the scope)
 - implemented by abstract base class **OperatorBase**
- operators store references to their successor operators (sub-operators)
 - build a graph structure (execution graph)

Operators

IOperator
Interface
↳ IConstrainedItem

[-] Properties

- Canceled { get; } : bool*
- Name { get; set; } : string*
- SubOperators { get; } : IList<IOperator>*
- VariableInfos { get; } : ICollection<IVariableInfo>*
- Variables { get; } : ICollection<IVariable>*

[-] Methods

- Abort() : void*
- AddSubOperator(IOperator op) : void*
- AddSubOperator(IOperator op, int index) : void*
- AddVariable(IVariable variable) : void*
- AddVariableInfo(IVariableInfo variableInfo) : void*
- Execute(IScope scope) : IOperation*
- GetVariable(string name) : IVariable*
- GetVariableInfo(string formalName) : IVariableInfo*
- GetVariableValue(string formalName, IScope scope, bool recursiveLookup) : IItem*
- GetVariableValue(string formalName, IScope scope, bool recursiveLookup, bool throwOnError) : IItem*
- GetVariableValue<T>(string formalName, IScope scope, bool recursiveLookup) : T*
- GetVariableValue<T>(string formalName, IScope scope, bool recursiveLookup, bool throwOnError) : T*
- RemoveSubOperator(int index) : void*
- RemoveVariable(string name) : void*
- RemoveVariableInfo(string formalName) : void*

[+] Events

Variable Infos

- operators have to be independent of concrete variable names
- for example, functionality of a counter operator:
 - get variable from scope
 - increase the variable's value by 1
 - write new variable value back into the scope
- mechanism needed to translate formal variable names into actual variable names
 - formal variable name is used by the operator to identify a variable
 - actual variable name is the real name of the variable in the scope
- interface **IVariableInfo**
 - stores meta-information about variables
 - actual name, formal name, data type, description, kind, ...
 - used by operators to define which variables are required
- each operator has to define a variable info object for each variable it requires
 - usually done in the operator's constructor
- actual names can be set by the user when assembling an algorithm
- formal/actual name translation is automatically performed when using an operators GetVariableValue method



Example: Counter

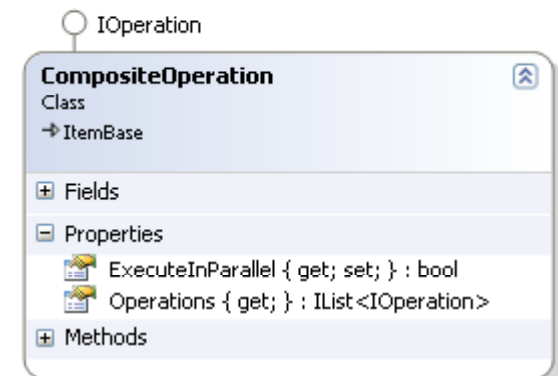
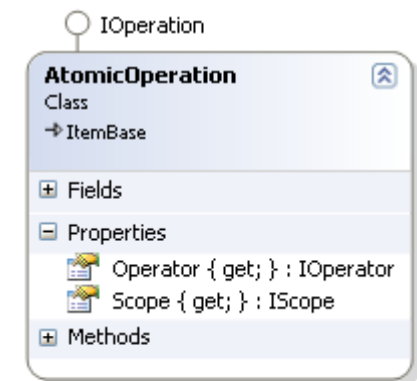
```
public class Counter : OperatorBase {
    public Counter() {
        AddVariableInfo(
            new VariableInfo("Value", // formal name
                            "Counter value", // description
                            typeof(IntData), // data type
                            VariableKind.In | VariableKind.Out) // kind
        );
    }

    public override IOperation Apply(IScope scope) {
        // perform formal/actual name translation
        IntData value = GetVariableValue<IntData>("Value", scope, true);

        value.Data++;
        return null;
    }
}
```

Operations

- operators store references to their sub-operators
- after execution operators have to decide which operators are executed next
- to enable step-wise execution and parallelism operators should not directly execute their successor operators
- operators that should be executed next and the corresponding scopes are represented as operations and returned to the engine
- interface **IOperation**
 - marker interface inherited from `IItem`
 - doesn't contain anything
 - implemented by **AtomicOperation** and **CompositeOperation** (composite pattern)
- class **AtomicOperation**
 - represents a single operation
 - contains an operator (**Operator**) and a scope (**Scope**) on which the operator should be executed
- class **CompositeOperation**
 - represents an operation consisting of several other operations
 - contains a collection of sub-operations (**Operations**)
 - contains flag to indicate if sub-operations can be executed in parallel (**ExecuteInParallel**)



Example: ConditionalBranch

```
public class ConditionalBranch : OperatorBase {
    public ConditionalBranch() {
        AddVariableInfo(new VariableInfo("Condition",
            "A boolean variable that decides the branch",
            typeof(BoolData),
            VariableKind.In));
    }

    public override IOperation Apply(IScope scope) {
        BoolData resultData = GetVariableValue<BoolData>("Condition", scope, true);
        bool result = resultData.Data;

        if (result)
            return new AtomicOperation(SubOperators[0], scope); // true branch
        else
            return new AtomicOperation(SubOperators[1], scope); // false branch
        return null;
    }
}
```

Example: SequentialProcessor, UniformParallelSubScopesProcessor

```
public class SequentialProcessor : OperatorBase {
    public override IOperation Apply(IScope scope) {
        CompositeOperation next = new CompositeOperation();
        for (int i = 0; i < SubOperators.Count; i++)
            next.AddOperation(new AtomicOperation(SubOperators[i], scope));
        return next;
    }
}
```

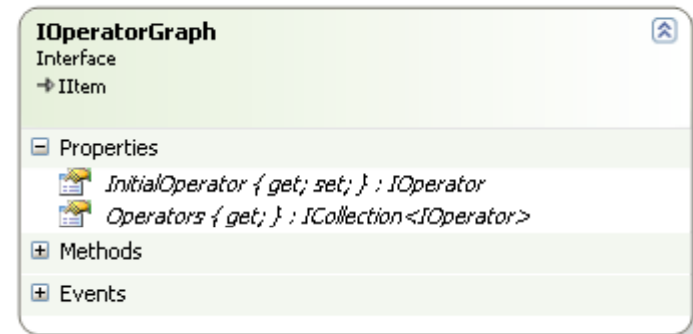
```
public class UniformParallelSubScopesProcessor : OperatorBase {
    public override IOperation Apply(IScope scope) {
        CompositeOperation next = new CompositeOperation();
        next.ExecuteInParallel = true;

        for (int i = 0; i < scope.SubScopes.Count; i++)
            next.AddOperation(new AtomicOperation(SubOperators[0], scope.SubScopes[i]));

        return next;
    }
}
```



Engines

- operators must not call their successor operators directly
- engine is responsible for operator execution
- engines are able to deal with parallelism
- interface **IOperatorGraph**
 - represents an algorithm (a graph of operators)
 - contains an initial operator (**InitialOperator**) and a list of all operators (**Operators**)
 - implemented by **OperatorGraph**
- interface **IEngine**
 - handles execution of operator graphs
 - contains an operator graph (**OperatorGraph**) and a global scope (**GlobalScope**)
 - defines methods to start, stop or abort execution
 - implemented by **SequentialEngine** and **ThreadParallelEngine**
- each execution start with an empty global scope



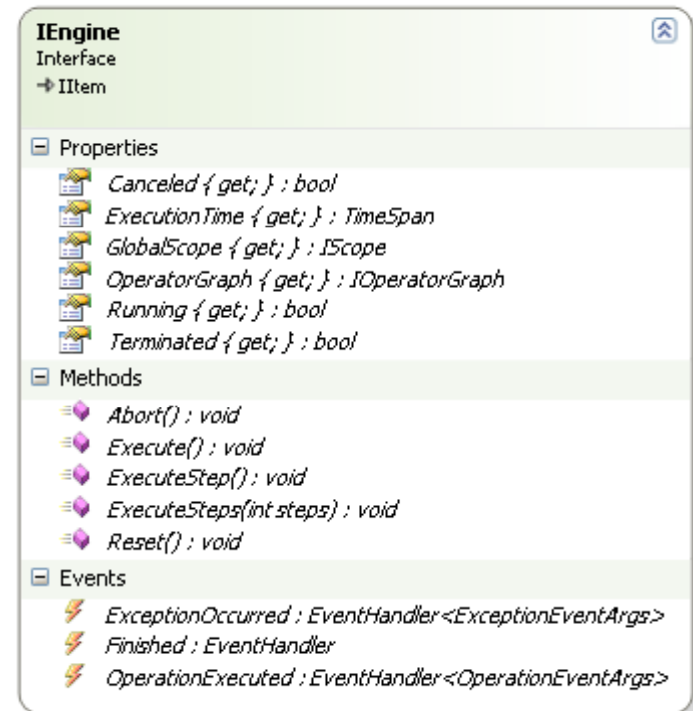
IOperatorGraph
Interface
→ IItem

[-] Properties

-  *InitialOperator { get; set; } : IOperator*
-  *Operators { get; } : ICollection<IOperator>*







[+] Methods

[+] Events








IEngine
Interface
→ IItem




[-] Properties

-  *Canceled { get; } : bool*
-  *ExecutionTime { get; } : TimeSpan*
-  *GlobalScope { get; } : IScope*
-  *OperatorGraph { get; } : IOperatorGraph*
-  *Running { get; } : bool*
-  *Terminated { get; } : bool*

[-] Methods

-  *Abort() : void*
-  *Execute() : void*
-  *ExecuteStep() : void*
-  *ExecuteSteps(int steps) : void*
-  *Reset() : void*

[-] Events

-  *ExceptionOccurred : EventHandler<ExceptionEventArgs>*
-  *Finished : EventHandler*
-  *OperationExecuted : EventHandler<OperationEventArgs>*

Demonstration