

HeuristicLab 2.0

Optimization API

HeuristicLab Mission Statement

- HeuristicLab ...
 - is an environment for heuristic optimization
 - is suitable for scientific algorithm development, university lectures, industry projects
 - is generic and extensible
 - is comfortable and easy to use
 - based on state-of-the-art APIs
 - uses modern software engineering concepts

HeuristicLab 1.0 – History

- Summer 2002
 - starting shot of the development of HeuristicLab
- 27. Dec. 2002
 - first inofficial release of HeuristicLab 1.0.0
- 11. Mar. 2003
 - birth of SASEGASA
- 26. Aug. 2003
 - beginning of the development of HeuristicLab Grid
- 08. Sep. 2004
 - presentation of "HeuristicLab Grid" at ICSS 2004
- 04. Nov. 2004
 - HeuristicLab Homepage goes online
- 18. Nov. 2004
 - release of HeuristicLab 1.0.0 (37 plugins)
- 22. Nov. 2004
 - acceptance of the paper "HeuristicLab" at ICANNGA 2005
- 20. May 2005
 - starting shot of the development of HeuristicLab 2.0
- 19. Aug. 2005
 - release of HeuristicLab 1.1.0 (45 plugins)

HeuristicLab 1.0 – Features

- paradigm independency
 - open for different kinds of heuristic optimization algorithms and problems
- extensibility
 - separation between algorithms and problems
 - plugin concept
- comfort
 - GUI, setup, API documentation

HeuristicLab 1.0 – Drawbacks

- very high level of abstraction
 - no specific APIs (e.g. for GAs, GP, local search)
 - plugins are monolithic
 - loads of duplicated code
- strict separation between algorithms and problems
 - too strict for several applications (e.g. TS)
 - no "clean" way for hybridization
- limited extensibility
 - plugins cannot extend other plugins
 - plugins cannot embed functionality directly into the main window
 - different file formats and editors are not supported
 - no integration of other modules (e.g. solution analyzer, Hive, ...)
- complicated algorithm development
 - even small changes in algorithms require rather deep knowledge about HeuristicLab and quite good programming skills

HeuristicLab 1.0 – Drawbacks

- limited comfort
 - execution of algorithms cannot be interrupted
 - algorithm runs cannot be saved and restored
 - implementation of GUI elements is quite time-consuming
 - binary and soap serialization are not suitable to store problems
 - statistical and graphical analyses are rather complicated
 - no support concerning experiment planning
 - distributed computing is not integrated
 - TestBench is not integrated
- no multi-objective optimization
- based on MS .Net 1.1
 - MS .Net 2.0 offers lots of new features

HeuristicLab 1.0: Quite cool, but ...

we are computer scientists, so ...

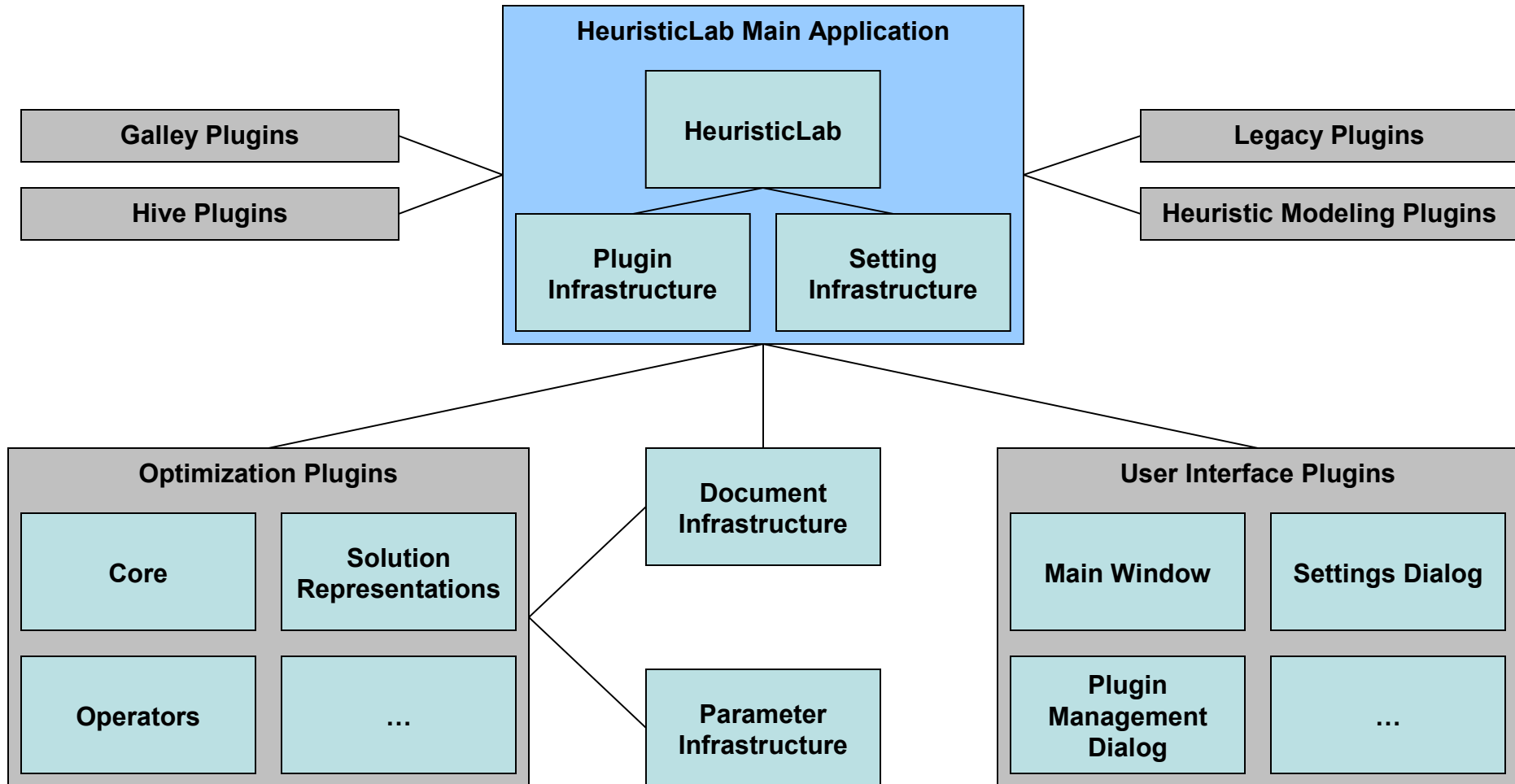
CAN'T WE DO BETTER?

WELL, LET'S SEE...

HeuristicLab 2.0 – Requirements

- everything is based on plugins
 - HeuristicLab is a network of interacting plugins
 - plugins are not included in the setup but installed from update locations
 - customization
- modular design
 - interaction between objects is based on interfaces
 - everything should be exchangeable
- maximize comfort for programmers
 - generic APIs for algorithms, parameters, problems, documents, charting, statistical analyses
- maximize comfort for users
 - algorithm runs can be interrupted, saved, loaded, etc.
 - experiment planning
 - reduce need for programming skills
- full integration
 - provide a single platform
 - integrate HeuristicModeller, Galley, Hive, ...

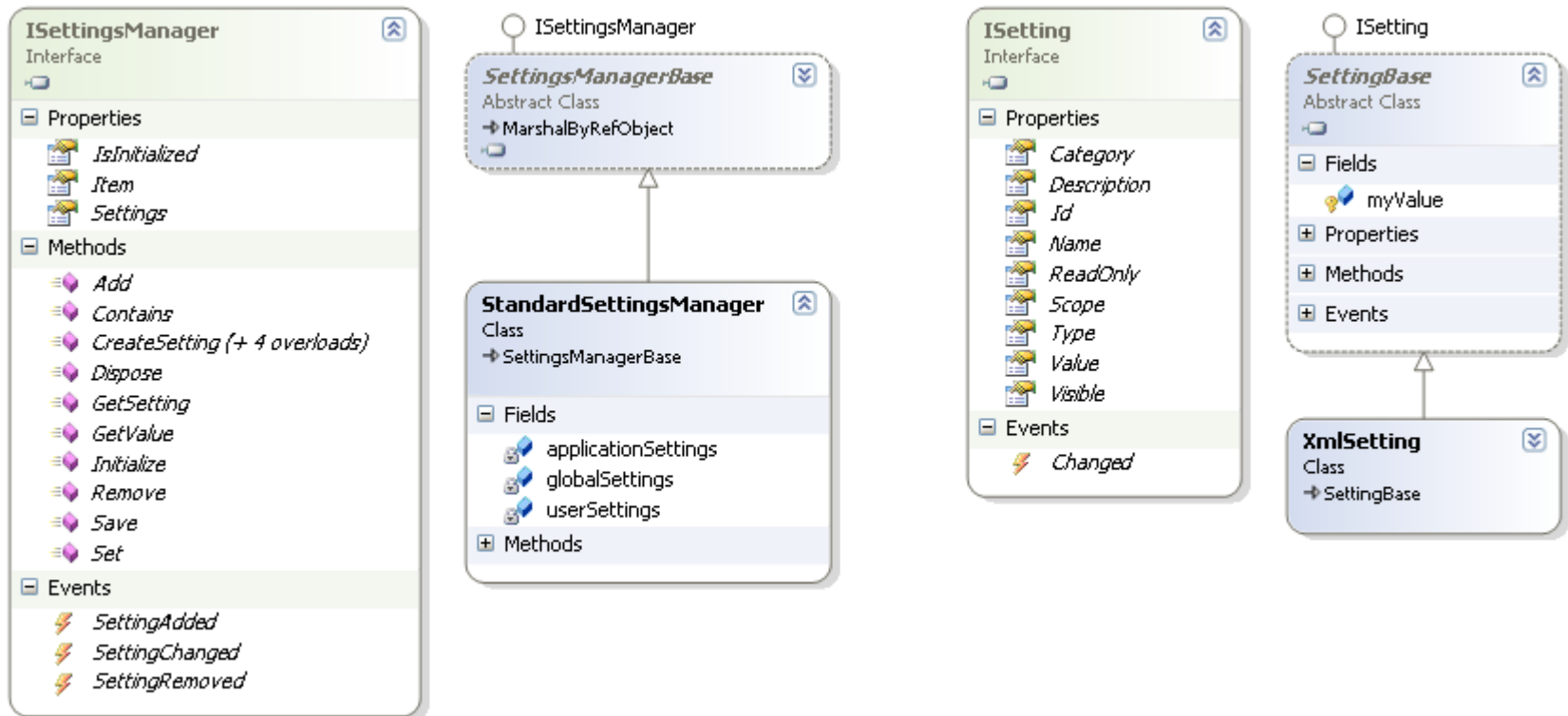
Architecture Overview



Setting Infrastructure

- generic API to manage settings
 - application settings
 - plugin settings
 - different scopes (global, application, user)
 - events notify about setting changes
- flexibility
 - every XML-serializable object can be stored
- modular design
 - access via interfaces
 - back-end can be exchanged
 - save settings in XML files, data base, ...
- ease of use
 - ISettingsManager is used to add, access, remove settings

Setting Infrastructure



Plugin Infrastructure

- generic API to handle plugins
 - different categories (ClassLibrary, Service, Runnable, ...)
 - different status (Available, Installed, Loaded)
- plugin management
 - plugin manager is used to install, initialize, update, dispose, remove plugins
 - events notify about every action
 - coupling via extension points
- plugin information
 - contains id, name, description, plugin type, category, status, ...
 - information about authors, files, extensions
 - directly accessible via API
 - back-end can be exchanged
 - save plugin description in XML files, data base, ...
- plugin class
 - must be provided by every plugin
 - callbacks for installation, initialization, update, disposal, removal
- modular design

Plugin Infrastructure

IPluginManager
Interface

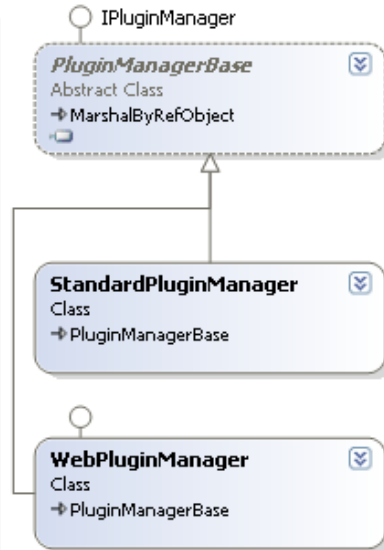
Properties

- IsInitialized
- PluginInfos
- Plugins

Methods

- Dispose
- DisposePlugin
- GetAvailablePlugin (+ 1 overload)
- GetAvailablePlugins (+ 1 overload)
- GetExtendingPlugins (+ 1 overload)
- GetExtensionPoints (+ 1 overload)
- GetExtensions (+ 1 overload)
- GetObject
- GetPlugin (+ 1 overload)
- GetPluginInfo
- Initialize
- InitializePlugin
- InstallPlugin
- PauseService
- RealizeExtension<T>
- RemoveObject
- RemovePlugin
- ResumeService
- Run
- RunPlugin (+ 1 overload)
- StartService
- StopService
- StoreObject
- UpdatePlugin

Events



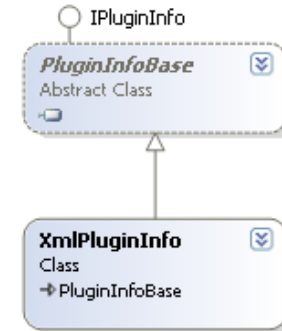
IPluginInfo
Interface

Properties

- Authors
- Copyright
- Description
- ExtensionPoints
- Extensions
- Files
- Id
- Name
- PluginCategory
- PluginStatus
- PluginType
- RequiredPlugins
- StartupAction
- UpdateLocation
- Version

Methods

- Extends (+ 1 overload)
- GetExtensions (+ 1 overload)



IPlugin
Interface

Properties

- Info

Methods

- Dispose
- Initialize
- Install
- PrepareForUpdate
- Remove
- Update

PluginBase
Abstract Class
↳ MarshalByRefObject

Plugin Infrastructure

- extension points and extensions
 - inspired by the Eclipse plugin concept
 - coupling between plugins without referencing
 - specified in plugin description
- three main actors
 - host plugin
 - defines extension point (id, name, description)
 - defines extension point interface
 - client plugin
 - defines extension (id, name, description)
 - specifies extended extension point (id)
 - provides type implementing extension point interface
 - provides meta information
 - plugin manager
 - provides information about extending plugins
 - used to realize extensions (object creation)

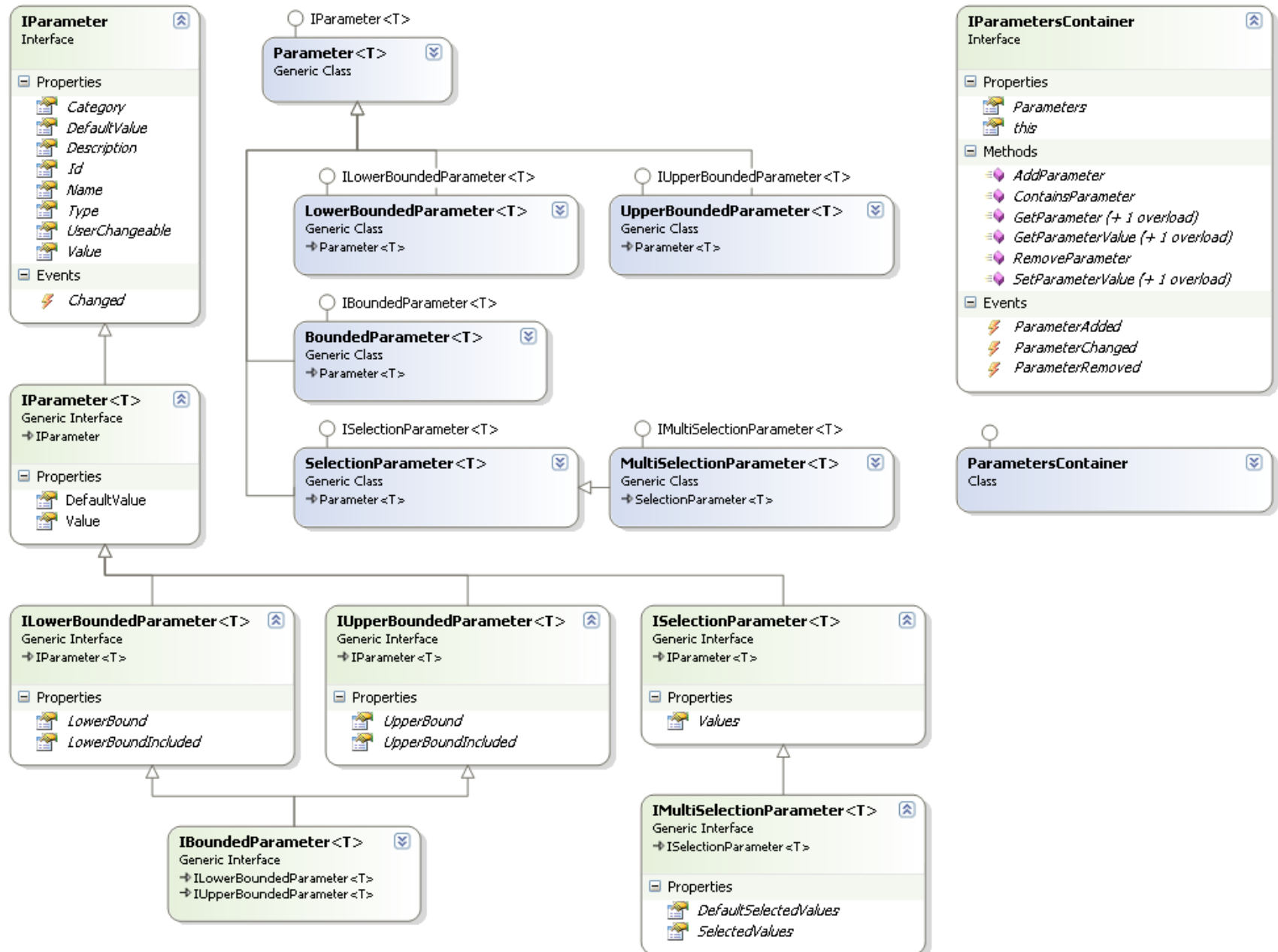
Plugin Infrastructure

```
IPluginManager manager = PluginManager.Manager;  
IPluginInfo info = manager.GetPluginInfo("ID");  
foreach (IExtensionInfo extension in  
    info.GetExtensions("HeuristicLab.Plugins.Optimization.Operators")) {  
    IOperator op = manager.RealizeExtension<IOperator>(extension);  
    op.Apply(...);  
    ...  
}
```

Parameter Infrastructure

- generic API for algorithm and problem parameters
 - different types of parameters
 - bounded, selection, multi-selection
 - parameters are grouped into categories
 - events notify about parameter changes
- flexibility
 - parameters are typed (generics)
 - every kind of object can be used as a parameter
- modular design
 - based on interfaces
 - plugins can implement own parameter types
- ease of use
 - IParametersContainer is used add, access, remove parameters
 - single front-end can be used to access parameters from the GUI

Parameter Infrastructure



Parameter Infrastructure

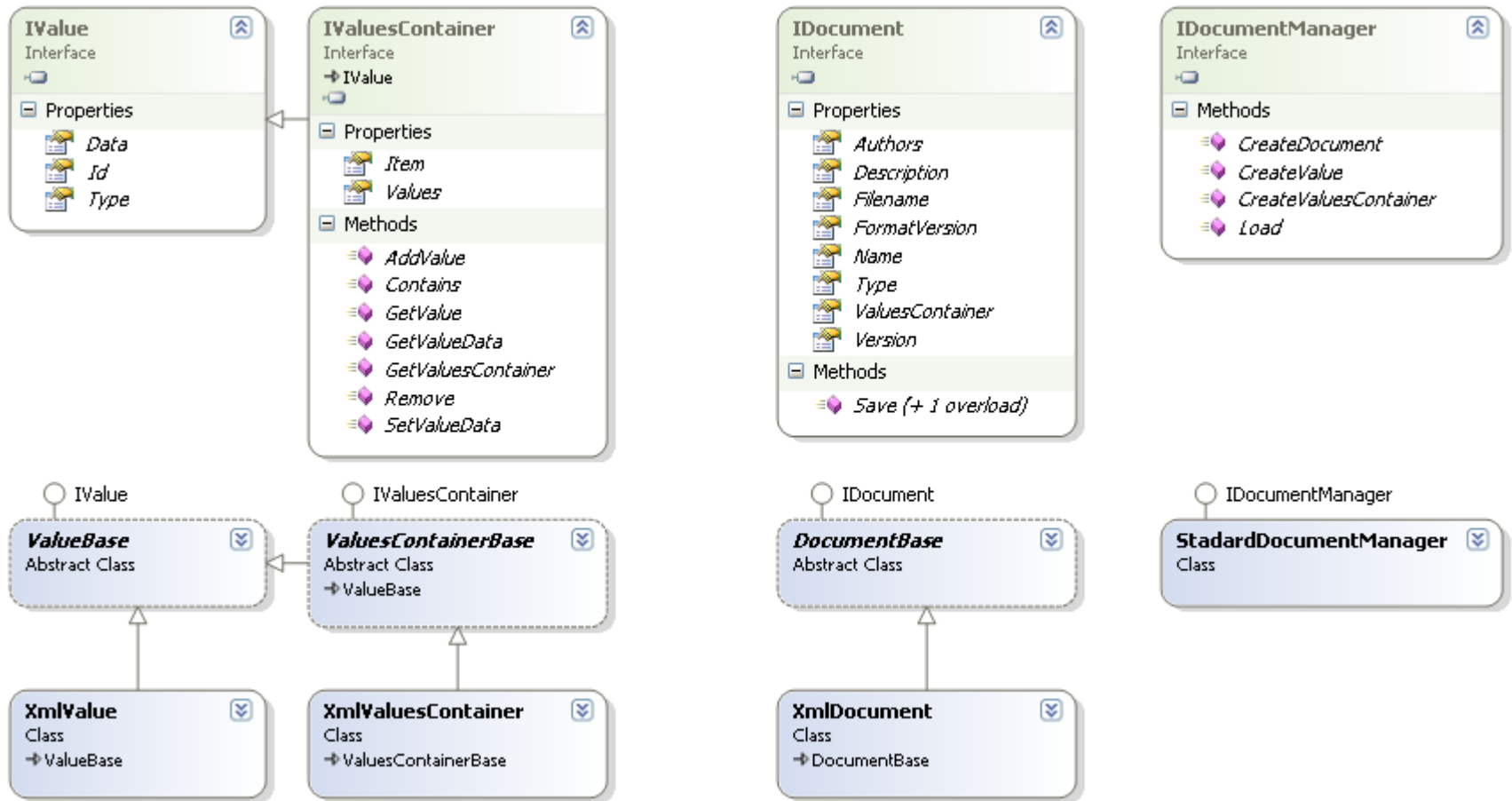
```
IParametersContainer container;  
IParameter parameter = new Parameter<IOperator>("ID", "Name", "Description", "Category");  
container.AddParameter(parameter);
```

```
IParametersContainer container;  
container.SetParameterValue<IOperator>("ID", value);  
container.GetParameterValue<IOperator>("ID");
```

Document Infrastructure

- generic API for documents
 - document contains meta-information (version, type, authors, ...)
 - document contains arbitrary values
 - value containers group values
 - value containers are values themselves
- flexibility
 - values may be of any (XML-serializable) type
- modular design
 - based on interfaces
 - own document and value types can be implemented
- ease of use
 - IDocumentManager is used to save, load documents and to create new values and value containers
- integration (still to do)
 - document infrastructure is integrated into main window
 - plugins provide viewers/editors to work with documents

Document Infrastructure



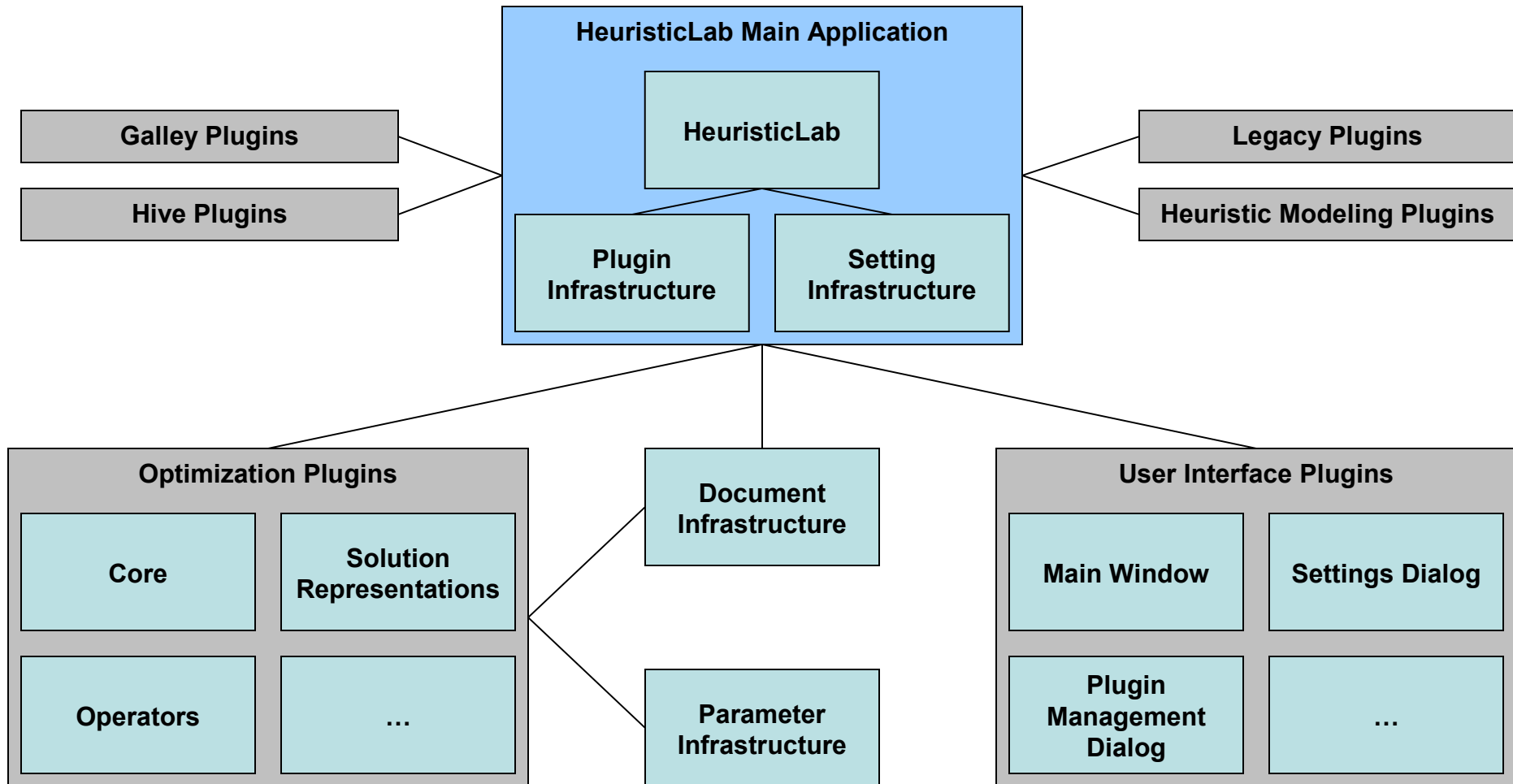
Document Infrastructure

```
IDocument document;  
document = DocumentManager.Manager.CreateDocument("Type", version);  
document.Filename = filename;  
document.ValuesContainer.AddValue(...);  
document.Save();
```

```
IDocument document;  
document = DocumentManager.Manager.Load(filename);  
object value = document.ValuesContainer.GetValueData("ID");  
...
```

```
IValuesContainer container;  
container = DocumentManager.Manager.CreateValuesContainer("ID");  
IValue value;  
value = DocumentManager.Manager.CreateValue("ID", value);  
container.AddValue(value);  
...
```

Architecture Overview



Optimization Plugins

- main ideas
 - different elements interact in an heuristic optimization process
 - solution encodings, fitness functions, solution generators
 - operators (mutation, neighborhood, crossover, ...)
 - splitting, replacement
 - termination criterions
 - algorithms, problems
 - ...
 - plugins contain elements
 - plugin infrastructure provides discovery mechanism
 - elements are identified by extensions
 - elements can use other elements
 - elements form a tree (or more general a graph)
 - algorithms are just a successive application of operators
 - operators manipulate one or more solutions
 - elements can be saved and restored
 - internal state has to be saved
 - consistency has to be guaranteed

Optimization Elements

- everything (except solutions) is an optimization element
- elements provide basic properties
 - `IOptimizationElement`
 - `string Name`
 - `IParametersContainer Parameters`
 - `IParametersContainer Status`
- elements provide a control for the GUI
 - `IOptimizationElementControl Control`
- elements provide a method to clone themselves
 - `IOptimizationElement Copy()`
- elements provide a mechanism to save and restore themselves
 - `IValuesContainer GetValues(string id)`
 - `static OptimizationElementBase Create(IValuesContainer values)`

Optimization Elements

- elements build a tree structure
 - `IOptimizationElement Parent`
 - `IList<IOptimizationElement> Children`
 - `IOptimizationWorkbench Workbench`
- elements provide a mechanism to add or remove them from the tree
 - `void Attach(IOptimizationElement parent)`
 - `event EventHandler Attached`
 - `void Detach(IOptimizationElement parent)`
 - `event EventHandler Detached`
- elements have a check mechanism to ensure valid settings
 - `bool Check(IList<string> messages)`
- elements can be initialized
 - `void Initialize()`
- elements realize a tree-based 2-phase commit to ensure consistency
 - `void RequestCommit()`
 - `void Commit()`
 - `void Rollback()`

Solutions

- solutions contain data and quality value(s)
 - ISolution
 - `double Quality`
 - `double[] Qualities`
 - `object Data`
- solutions provide a method to clone themselves
 - `ISolution Copy()`
- solutions provide a mechanism to save and restore themselves
 - `IValuesContainer GetValues(string id)`
 - `static Solution Create(IValuesContainer values)`
- specific solutions redefine data property
 - IPermutationSolution
 - `new int[] Data`

Solutions

- solution representations are used to generate data
 - ISolutionRepresentation
 - `object CreateSolutionData(params object[] list)`
 - `object CreateRandomSolutionData(params object[] list)`
- solutions are compared by comparison operators
 - IComparisonOperator
 - inherited from IComparer
- solution generators are used to create new solutions
 - ISolutionGenerator
 - `ISolution Generate()`

Operators

- operators manipulate solution arrays
 - IOperator
 - `ISolution[] Apply(ISolution[] solutions)`
- operators manipulate single solutions
 - ISolutionProcessingOperator
 - `void Apply(ISolution solution)`
- operators merge solution arrays
 - IMergeOperator
 - `ISolution[] Apply(ISolution[][] solutions)`
- operators split solution arrays
 - ISplitOperator
 - `ISolution[][] Apply(ISolution[] solutions)`

Conditions

- termination criterions are represented as conditions
 - ICondition
 - `bool Check()`
- conditions are also used in loop and branch operators
- arbitrary conditions can be combined
 - AndConjunctionCondition
 - OrConjunctionCondition

Algorithms

- algorithms contain basic status properties
 - IAlgorithm
 - `bool Terminated`
 - `bool Interrupted`
 - `TimeSpan ComputationTime`
 - `long EvaluatedSolutions`
- algorithms control the workflow of the optimization process
 - `void ExecuteStep`
 - `void Execute`
 - `void Interrupt`
- algorithms contain processed solutions
- algorithms call operators to manipulate solutions

Optimization Workbenches

- workbenches integrate all required elements
 - IOptimizationWorkbench
 - `bool FixRandomSeed`
 - `int RandomSeed`
 - `Random Random`
 - `ISolutionRepresentation SolutionRepresentation`
 - `IProblem Problem`
 - `IAlgorithm Algorithm`
- workbenches provide a discovery mechanism to get elements
 - `IList<T> GetAvailableOptimizationElements<T>()`
 - `IList<IOptimizationElement> GetAvailableOptimizationElements(Type type)`
- workbenches hide threading
 - `void Start()`
 - `void Stop()`
 - `event ExceptionEventHandler ExceptionOccurred`
 - `event EventHandler Stopped`

Optimization Plugins Namespaces

- HeuristicLab.Plugins.Optimization
 - Interfaces
 - Core
 - SolutionRepresentations
 - Binary
 - IntArray
 - Permutation
 - Real
 - Operators
 - Basic
 - Concurrent
 - Control
 - Replacement
 - Reporting
 - Selection

Optimization Plugins Namespaces

- HeuristicLab.Plugins.Optimization
 - UI
 - OptimizationWorkbench
 - StandardOptimizationWorkbench
 - SimpleOptimizationWorkbench
 - Routing
 - TravelingSalesmanProblem
 - Evolutionary
 - Core
 - OffspringSelection
 - HeuristicModeling
 - ...

Discussion

? **?** **?**