jInfer TwoStep simplifier design and implementation

Michal Klempa, Mário Mikula, Robert Smetana, Michal Švirec, Matej Vitásek
Advisors: RNDr. Irena Mlýnková, Ph.D., Martin Nečaský, Ph.D.

Praha, 2011

Target audience: developers willing to extend jInfer, scientist willing to implement own inference methods
*Note: we use the term **inference** for the act of creation of schema throughout this and other jInfer documents.*

Responsible developer:	anti
Required tokens:	cz.cuni.mff.ksi.jinfer.base.interfaces.RuleDisplayer
Provided tokens:	cz.cuni.mff.ksi.jinfer.base.interfaces.inference.Simplifier
Module dependencies:	AutoEditor, Base, JUNG, Lookup
Public packages:	

1 Using of factory pattern

TwoStepSimplifier is divided into three submodules, which are then divided into submodules and so on. Each submodule is simply class, properly anotated. User selects proper classes - submodules to work in chain on inference. Classes are then, in runtime looked up by using NetBeans lookup mechanism (see [KMS+, p. 16]). But NetBeans holds one instance of each class, that may be looked upon. When user runs inference first time, classes are looked up and used. Maybe class members are initialised, instances are in some state after inference completes. Then, user clicks run button again, and same instances of classes are returned by lookups. Oops, theese are not freshly created instances, using them as they are may cause harm. The problem is ilustrated on fig. 1(a).

There are basically two solutions:

- force each class (submodule) that is being looked up in inference process, to implement some sort of cleanup method, that would restart it into fresh state and make it ready to use by another inference run,
- or create new instances of theese classes in each inference run and make singleton classes (those which NetBeans return by lookups) their factory classes.

We decided for the latter approach (as ilustrated on fig. 1(b) It enables us not only to produce fresh class instance in each inference run, but also factory classes implement obligatory module methods such as `getName`, `getDescription` and so on, which are really same in each inference run. Each submodule is then defined by (at least) two interfaces. One is the factory interface, like this one:

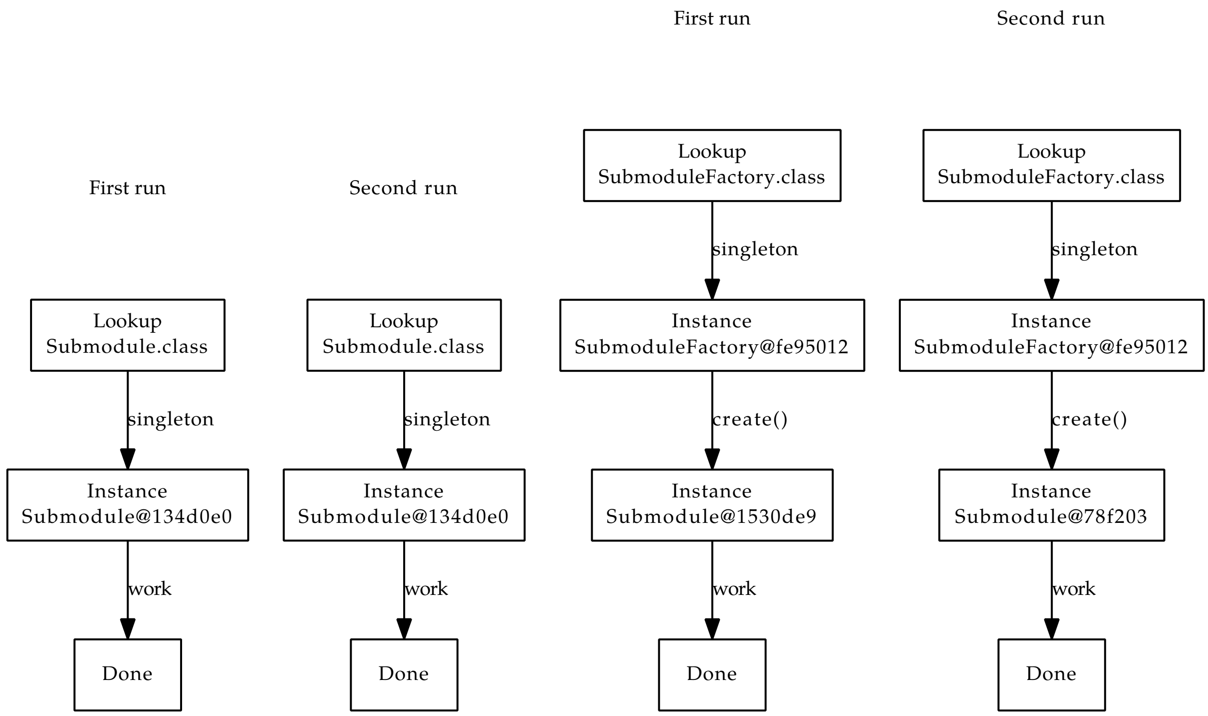
```
public interface AutomatonSimplifierFactory extends
                    NamedModule, Capabilities, UserModuleDescription {
  <T> AutomatonSimplifier<T> create();
}
```

It extends `NamedModule` with its modules describing methods:

```
String getName();
String getDisplayName();
String getModuleDescription();
```

It extends `Capabilities`, thus each module have to answer, if it has some capabilities. And, in TwoStep we often extend `UserModuleDescription`, which defines method:

```
String getUserModuleDescription();
```



(a) Same instance of class returned by NetBeans in two successive inference runs

(b) Factory solution of singleton lookup class problem

It returns description of the module, comprehensive to user, which is then displayed in properties panels.

In some circumstances, it is useful to have method `create()` generic. The *`AutomatonSimplifier`* works with `Automaton`, which itself is generic too. But simplifying does not depend on type of symbol of automaton, so interface `AutomatonSimplifier<T>` is also generic as simplifier can simplify automaton of any java type symbol. Factory interface deals with this by defining `create()` method generic too.

Second interface is the one, which is returned by `create()` method. That is the interface, which defines real work cycle with submodule (in example it is `AutomatonSimplifier<T>`). We call this interface the worker interface of submodule. Usage of this factory pattern follows the routine:

```
final Properties p = RunningProject.getActiveProjectProps(getName());

AutomatonSimplifierFactory f = ModuleSelectionHelper.lookupImpl(AutomatonSimplifierFactory.class,
   p.getProperty(PROPERTIES_AUTOMATON_SIMPLIFIER));

AutomatonSimplifier<AbstractStructuralNode> autSmp = f.<AbstractStructuralNode>create();
...
give some work to autSmp
```

If our module has some submodules, we often implement lookups for submodules implementations in our own factory create method. Worker class receives factories of all submodules it needs as a constructor parameters.

Lets look on `AutomatonMergingStateFactory`, that is factory of module, which has AutomatonSimplifier as submodule. Its create method looks like this (shortened):

```
@Override
public ClusterProcessor<AbstractStructuralNode> create() {
  LOG.debug("Creating new ClusterProcessorAutomatonMergingState.");
  return new AutomatonMergingState(getAutomatonSimplifierFactory(), getRegexpAutomatonSimplifierFactory())
}
```

Methods `getAutomatonSimplifierFactory` and `getRegexpAutomatonSimplifierFactory` are analogical, we show you the former one:

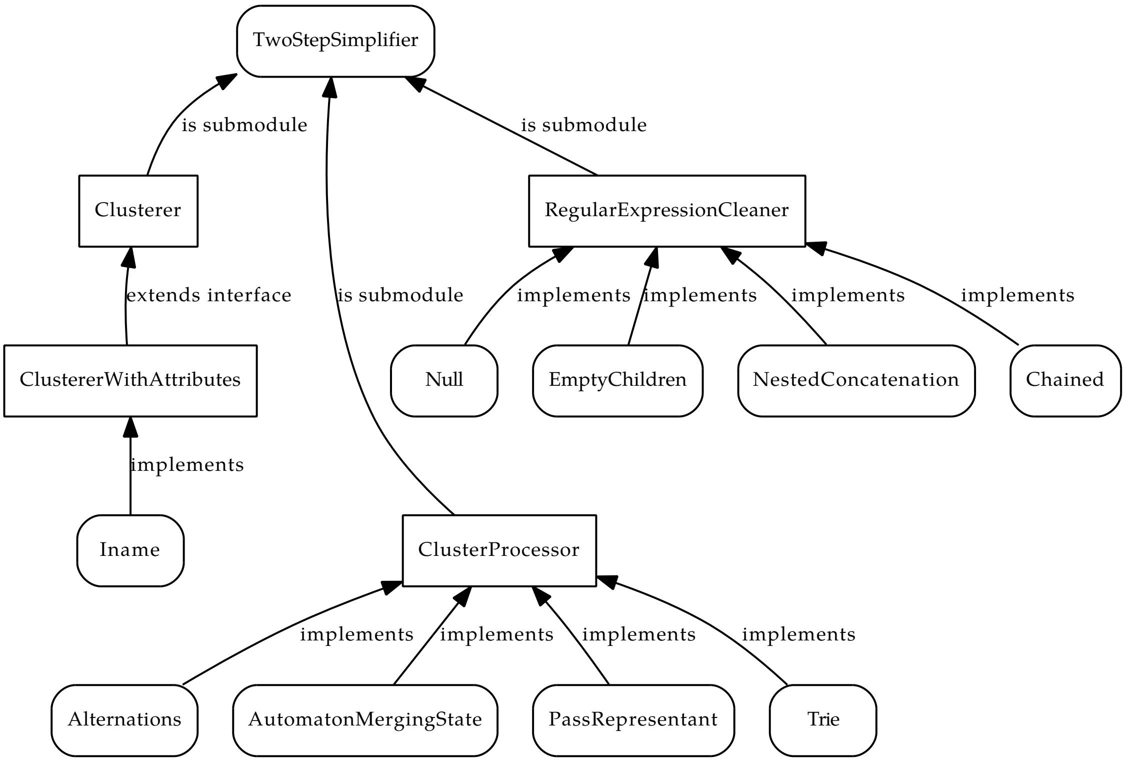


Figure 1: Submodules of TwoStep simplifier

```
private AutomatonSimplifierFactory getAutomatonSimplifierFactory() {
  final Properties p = RunningProject.getActiveProjectProps(getName());

  return ModuleSelectionHelper.lookupImpl(AutomatonSimplifierFactory.class,
          p.getProperty(PROPERTIES_AUTOMATON_SIMPLIFIER));
}
```

Cluster processor `AutomatonMergingState` then receives factories of `AutomatonSimplifier` and `RegexpAutomatonSimplifier` submodules in its constructor. Cluster processor then may create as many instances of submodule classes as it needs (maybe simplifying more than one automaton). Thorough this document, we will mention only worker interface when describing submodules, since all factory interfaces are designed same way just described.

2 TwoStep module

TwoStepSimplifier is inspired by [VMP08] design. Inference proceeds in two steps:

1. clustering of element instances into clusters of (probably) same elements
2. infering regular expression for each element from examples of element contents taken from all elements in cluster

Task of clustering is dedicated to *Clusterer* submodule, and task of infering regular expression for each cluster is dedicated to *ClusterProcessor* submodule. We will examine both of them. There is also third submodule called *RegularExpressionCleaner* actually, its purpose is just to beatify output regular expressions, no inference logic is implemented there. Modules are drawn on fig. 1. We provide one *Clusterer* and 4 *ClusterProcessor* implementations. Each of those will be explained further in this document.

TwoStepSimplfier is implemented in package `cz.cuni.mff.ksi.jinfer.twostep`. Module is implemented in two classes: `TwoStepSimplifier` (main logic), `TwoStepSimplifierFactory` (lookups, interface to other modules). Factory class implements `Simplifier` interface (defined in package `cz.cuni.mff.ksi.jinfer.base.interfaces.inference`). Thus has main method called `start` which receives Initial Grammar in form of

```
final List<Element> grammar
```

Each grammar rule is represented as class `Element`, where element is left side of rule and it's `getSubnodes()` method returns regular expression, which is right side of rule. Second parameter is

```
final SimplifierCallback callback
```

Callback which should be called when work is done. On output, simplifier provides simplified grammar as a parameter into callback function:

```
void finished(final List<Element> grammar);
```

Thus simplifier receives initial grammar and returns simplified grammar, both represented same way - as elements.

`TwoStepSimplifierFactory` in method `start()` created new `TwoStepSimplifier` class instance, providing three factories of submodules to its constructor. Then it calls `simplify(initialGrammar)` method on this instance and its result passes as parameter to callback function. Thats all, the magic comes in `TwoStepSimplifier` class.

Method `TwoStepSimplifier.simplify()` does basically this:

```
// 1. cluster elements
final Clusterer<AbstractStructuralNode> clusterer= clustererFactory.create();
clusterer.addAll(initialGrammar);
clusterer.cluster();

// 2. prepare emtpy final grammar
final List<Element> finalGrammar= new LinkedList<Element>();

// 3. process rules
final ClusterProcessor<AbstractStructuralNode> processor =
             clusterProcessorFactory.create();

for (Cluster<AbstractStructuralNode> cluster : clusterer.getClusters()) {
  final AbstractStructuralNode node =
                         processor.processCluster(clusterer, cluster.getMembers());
  final RegularExpressionCleaner<AbstractStructuralNode> cleaner =
                         regularExpressionCleanerFactory.<AbstractStructuralNode>create();
  // 4. add to rules
  finalGrammar.add(
              new Element(node.getContext(),
              node.getName(),
              node.getMetadata(),
              cleaner.cleanRegularExpression(((Element) node).getSubnodes()),
              attList));
}

return finalGrammar;
```

It creates clusterer, gives it all rules, orders it to cluster elements. Then, empty list of elements is created as final (simplified) grammar. For each input rule in initialGrammar, submodule *`ClusterProcessor`* is called to do infering of regular expression of that element. Finally, regular expression cleaning is done in submodule and new `Element` instance is created as a copy of processed node, but with cleaned regexp.

Now we will examine submodules for clustering, processing and cleaning.

TODO sentinel cant be on left side TODO sentinel processing

2.1 Clusterer module

Lets start by example, let input document(s) contain XML:

```
<person name="john" surname="smith">
  <info>
    Some text
```

```
    <note/>
  </info>
</person>
<inform>
  Another text
  <note/>
</inform>
<person>
    <information>
      Some text
      <note/>
    </information>
</person>
```

We examine info-like named elements. If we cluster elements sipmly by their name, we get one cluster with info element, another cluster with inform element and another one with information element. As [VMP08] suggests, we should consider element content in clustering process. If we take care of element content, elements info, inform and information would look same (in means of node tree inside them - text node and element note). Elements information and info have even same context (inside person element).

We implement clustering in `cz.cuni.mff.ksi.jinfer.twostep.clustering` package. One cluster is represented by class `Cluster<T>` with `T` as type of clustered items. This class simply holds java set of member of cluster and one of the references is held also in `representant` member.

We provide `Clusterer` interface for classes to implement. Its purpose is to cluster bunch of elements (rules) on input, into bunch of cluster class instances (cluters) on output. It has methods `add()` and `addAll()` for adding items for clustering. Center is method `cluster()`, which does the clustering itself. As it may be time-consuming operation, method throws `InterruptedException`. Implementation should take care of checking whether thread is user interrupted (see [KMS+, p. 12]). After clustering, implementation should hold clusters in member, as it it will be further asked by calling method `T getRepresentantForItem(T item)`. Given item to this method, one can ask for representant of cluster, to which the item belongs. If no such cluster exists (item was not added for clustering before), we recommend you throwing an exception rather than returning `null`. Missing item will probably indicate error in algorithm rather than normal workflow. One can pull clusters from clusterer by calling `getClusters()` method.

Basic work usage of clusterer is:

```
Clusterer <T> c = new MyContextClusterer<T>();
c.addAll(initialGrammar);
c.cluster();
...
c.getClusters();
or
c.getRepresentantForItem(x);
```

2.1.1 ClustererWithAttributes extended interface

Maybe you noticed that whole clusterer interface and cluster class are generic. They may be used as design pattern not only for clustering elements in inference process. To address clustering of elements in more detail, we created `ClustererWithAttributes<T, S>` interface, which extends `Clusterer<T>` interface. It adds method `List<Cluster<S>> getAttributeClusters(T representant)`, implying that each representant of type `T` (that is representant of some main cluster) has some "attribute" clusters associated with it. Attribute clusters are of type `S` and can be retrieved by calling `getAttributeClusters(x)`.

Finally, we use this scheme to implement clusterer `Iname<AbstractStructuralNode, Attribute>` class, which takes `AbstractStructuralNode` classes to cluster as main, and `Attribute` classes as attributes. Clustering is done based on elements `getName` equality test (ignore case). For each rule (element), it is first clustered by finding cluster where representant has same name - or create new cluster with this element as representant. Then we process right side of this element rule (that are nodes from `getSubnodes`). Since right side of rule is always concatenation, we simply take `.getSubnodes().getTokens()` list and iterate through it. Each node on right side, is examined:

- if it is simple data throw it to `SimpleDataClusterer` class, which does nothing more, than holds all of simple data in one cluster. But in future it may be replaced to cluster simple data somehow, to obtain meaningful content models in schemas.

- if it is element and it is tagged as sentinel by metada - search main clusters to find cluster with representant of same `getName()`, or create new cluster with this sentinel as representant. From IGG, sentinels may be only on right sides of rules and since each element in schemas has to be defined, there must exist another element with same name, which is not sentinel (and maybe it will come to process in future). So there can't be cluster with only one sentinel element in it.
- do nothing otherwise, since it is element, that has to have its subnodes defined, and therefore it is element that is proper grammar rule and therefore it has to be somewhere in main initial grammar list, thus it is already processed or will be on schedule in clustering.

Attributes of element are processed through helper attribute clusterer, which is created for each element cluster. We have bunch of elements of same name in cluster and one attribute clusterer associated with this bunch. This attribute clusterer is given all attributes instances encountered in all elements that are in bunch. It clusters them by name (case insensitive).

2.2 ClusterProcessor submodule

Cluster processor takes rules of one cluster of element and somehow obtains regular expression for that set of elements. It returns rule - element with name set to desired name of element in schema (not all elements in cluster have to have same name, if advanced clustering scheme is used, then processor has to choose right name for resulting element) and with subnodes set to regular expression infered. It process attributes of all elements in cluster to obtain meaningful schema attribute specification and theese attributes has to attach to resulting element.

Worker interface itself is defined as follows:

```
public interface ClusterProcessor<T> {
  T processCluster(
          final Clusterer<T> clusterer,
          final List<T> rules
          ) throws InterruptedException;
}
```

Maybe you are asking, why cluster processor is given the clusterer instance. Rules themself contain information about which elements to process, but clusterer has more information about the topic. Clusterer can tell you representant for any element in whole input (not only those elements in rules, but also those that may be on right side of rules), clusterer (if it is with attributes) has information about attributes <of each cluster.

We will now shortly describe each cluster processor implementation we've got.

2.2.1 PassRepresentant

Simple example to read. For each cluster return its representant as a rule to be in schema. This has nothing to do with infering grammar, it is just proof of submodules concept. Input documents are not valid against this odd grammar. Do not use this in practice, just read the code to understand the bare minimum needed to implement submodule.

2.2.2 Alternations

This processor simply gets all right sides from elements in cluster, puts them in one big list and creates alternation regular expression with this list as children. That is, it creates one big rule with alternation of every positive example observed. No generalization is done at all.

2.2.3 Trie

This processor takes all rules in a cluster, treats them like strings and builds a prefix tree (a "trie") of them. More precisely, it takes the first rule and delares it to be a long branch (concatenation of tokens) in a newly created tree. After that, it adds the remaining rules one by one as branches like this: as long as it can follow an existing branch, it follows it. As soon as the newly added branch starts to differ, it "branches off" (creates an alternation at that point) the existing tree and hangs the rest of the newly added rule there. Repeating this process creates a prefix tree describing all the rules in the cluster.

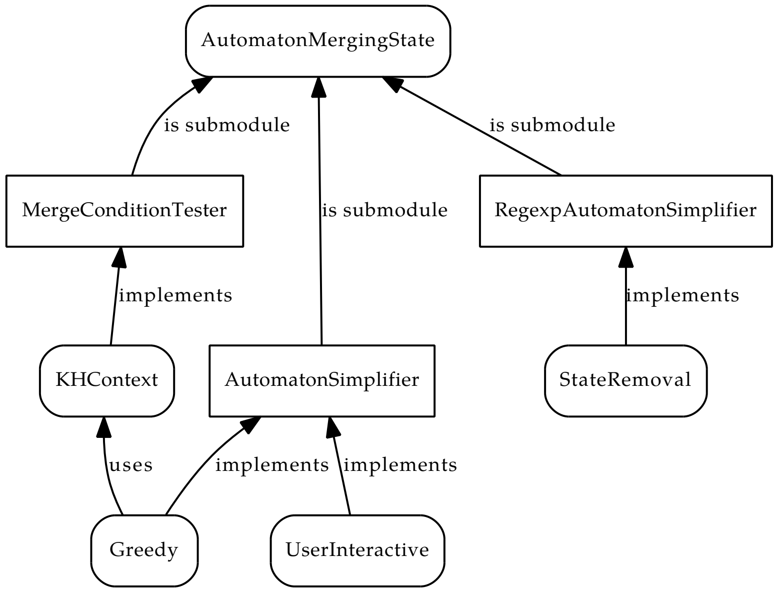


Figure 2: Submodules of AutomatonMergingState cluster processor

2.2.4 AutomatonMergingState

AutomatonMergingState is implementation of merging state algorithm on nondeterministic finite automaton. It creates prefix-tree automaton (PTA) from positive examples - right sides of rules given. Then it calls its submodule called *`AutomatonSimplifier`* to modify PTA to some generalized automaton by merging states.

Simplified automaton is then converted to an instance of `RegexpAutomaton` by using clone constructor in regexp automaton. Regexp automaton is automaton with regular expression as symbol on transitions. In automata theory, such automaton is called extended NFA. Clone constructing is done by converting each symbol in source automaton to regexp token with that symbol as content.

Regexp automaton is then passed into second submodule called *`RegexpAutomatonSimplifier`*. Its job is to derive regular expression from automaton, such that automaton and regular expression represents same language.

AutomatonMergingState has one more submodule, the *`MergeConditionTester`*, which is not called directly by AutomatonMergingState. It is at disposal for implementations of AutomatonSimplifier interface for testing, whether two states in automaton are equivalent and should be merged into one state.

AutomatonSimplifier implements solution searching logic in simplifying automaton by merging states. We implement greedy strategy in class `Greedy`. It simply asks given MergeConditionTester if it can merge any of automaton states and merges states until there are no states to be merged (on every pair of states asked, MergeConditionTester answers, they cannot be merged). One can implement ACO heuristics or MDL principle heuristics as AutomatonSimplifier submodule using MergeConditionTesters provided.

Whole submodule structure of AutomatonMergingState cluster processor is drawn on fig. 2. We have implemented *k,h-context* (see [Aho96]) state equivalence in class `KHContext`, which is used by `Greedy` to test mergability of states by default configuration.

2.2.5 StateRemoval

We are using state removal method (see [HW07]) to convert regexp automaton into equivalent regular expression. This is implemented in `StateRemoval` class (on fig. 3). We defined one submodule of this class with interface called `Orderer`. It has only one method to implement: `getStateToRemove`. Given automaton it has to return reference to one state which should be removed from automaton at first. State removal calls this submodule and removes states given until there are only two states in automaton - superInitial and superFinal states, with exactly one transition. That transition has final regular expression on it as symbol, it is read and returned to AutomatonMergingState. We implement one orderer, called `Weighted`. It is simple heuristic - weights all states (weight = sum of in | out | loop-transition regular expression lengths) and returns state with lowest weight.

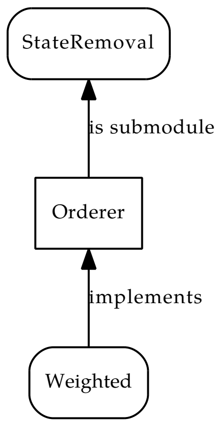


Figure 3: Modules of StateRemoval.

2.3 RegularExpressionCleaner module

Last we examine `RegularExpressionCleaner` interface. TODO anti cleaners

Whole TwoStep submodules structure is on fig. 4.

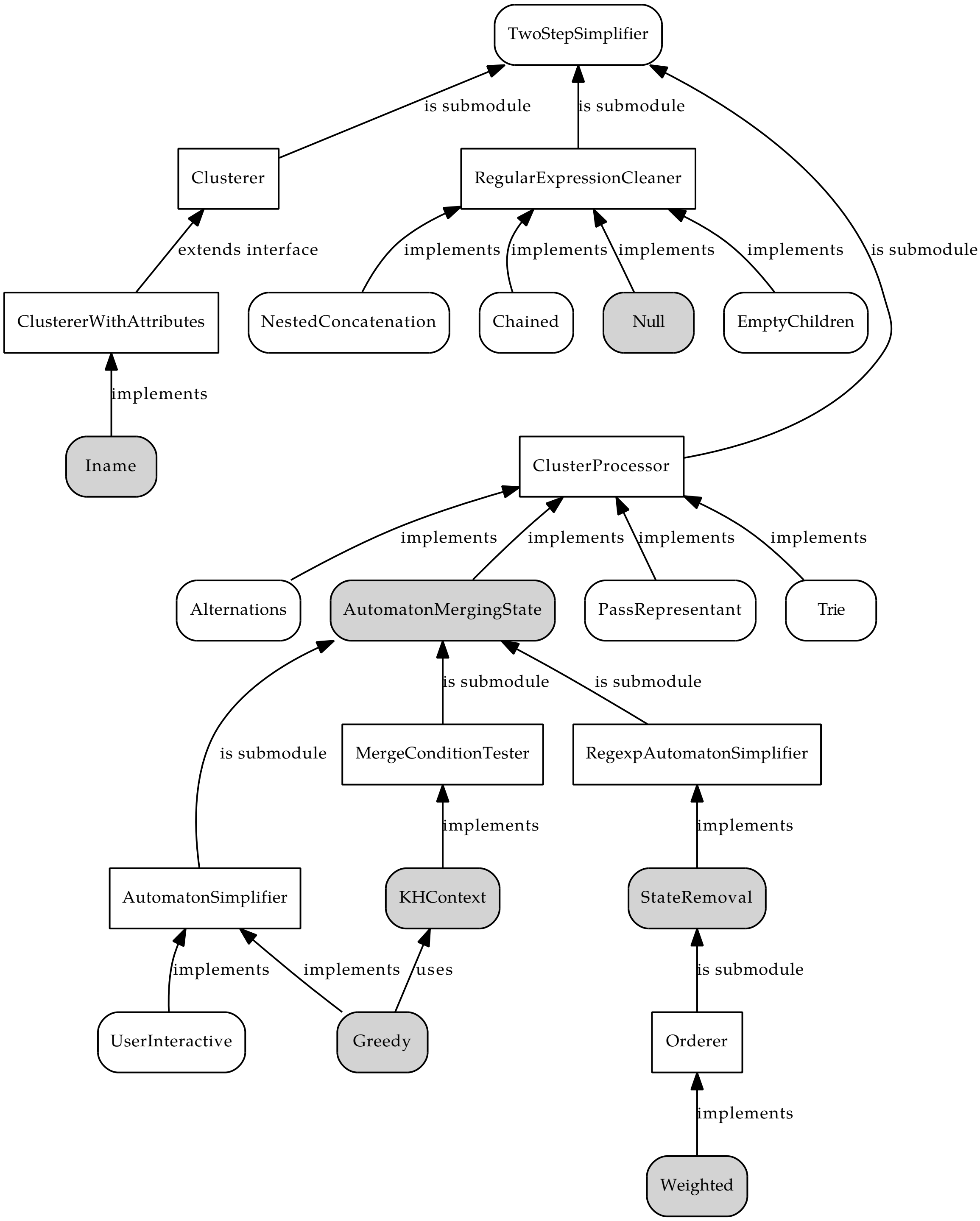


Figure 4: Modules of TwoStep simplifier and their submodules. Filled classes are default selection (best of).

References

[Aho96] H. Ahonen. *Generating grammars for structured documents using grammatical inference methods*. PhD thesis, Department of Computer Science, University of Helsinki, Series of Publications A, Report A-1996-4, 1996.

[HW07] Yo-Sub Han and Derick Wood. Obtaining shorter regular expressions from finite-state automata. *Theor. Comput. Sci.*, 370(1-3):110–120, 2007.

[KMS+] Michal Klempa, Mário Mikula, Robert Smetana, Michal Švirec, and Matej Vitásek. *jInfer Architecture*.

[VMP08] Ondřej Vošta, Irena Mlýnková, and Jaroslav Pokorný. Even an ant can create an xsd. In *DASFAA'08: Proceedings of the 13th international conference on Database systems for advanced applications*, pages 35–50, Berlin, Heidelberg, 2008. Springer-Verlag.