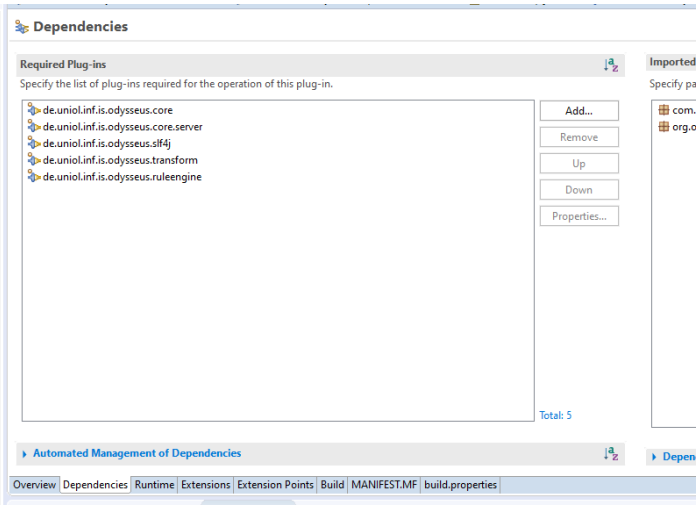
Creating new Operators

Typically, you will create an operator in an existing bundle. If you start with a new project from the scratch, you will need to create a new bundle.

To create an operator in your own bundle you need to add some dependencies to your Manifest.MF.

Open the MANIFEST.MF of your bundle and add the following dependencies:



To create a new operator from scratch, the following steps have to be followed:

1. Create a logical operator, which holds the necessary configuration parameters, e.g. a predicate for a filter or an attribute list for a projection. *ILogicalOperator* is the interface that needs to be implemented, *AbstractLogicalOperator* the base class that should be extended. **Important**: If the logical operator contains predicates, it must implement the interface *IProvidesPredicates*!
2. Create (at least) one physical operator implementation that is initialized by the information provided by the logical operator and can process the input data.
3. Optionally, create a rewrite rule. These rules are used to modify the logical query plan. In most cases this is the switch of operators, e.g. to push selections and projections close to the sources. If you create a new operator where the placement in the query plan does not matter. Provide rules to allow switching or else plans with the new operator may be optimizable.
4. Create a transformation rule. A transformation rule translates a logical operator into one of its physical counterparts.

Remark: This example adds an operator to an existing bundle. See Add a new Bundle and Feature if a new bundles should be created.

Important: See below if the output schema of the operator is different than the input schema.

Example: Route operator

In the following we will use the route operator to demonstrate how easy the integration of new operators in Odysseus is. The route operator is a more general form of the selection operator. It needs a number of predicates. If the first predicate evaluates on the incoming data to true, the input will be send to first output (port 0), if the seconds evaluates to true to the seconds output and finally, if no predicate evaluates to true to the last output port.

Step 1: Creating the logical operator

First, you need to decide if you want to place the operator in a new OSGi bundle or in an existing one. Independently of that, the operator must be placed in a package ending with *logicaloperator* if it should be integrated automatically.
To create your own logical operator you need to extend *AbstractLogicalOperator* (which implements *ILogicalOperator*). For convenience reasons there are base implementations for operators with single input (*UnaryLogicalOp*) and binary input (*BinaryLogicalOp*). As a naming convention, the class name should end with *AO* (for algebra operator).
This class must provide (at least) two constructors and the clone method. The default constructor is required as instances of logical operators are created by *newInstance()*. **Clone must call the copy constructor and the copy constructor must call the super copy constructor!** If at runtime an error like "has no owner" is called, in most cases this is because of the missing call to the super copy constructor.
Finally, the class needs setters and getters for the parameter it should keep

RouteAO

```
/********************************************************************************
 * Copyright 2011 The Odysseus Team
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
```

```
 * You may obtain a copy of the License at
 *
 *     http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */
package de.uniol.inf.is.odysseus.tutorial.logicaloperator;

import java.util.LinkedList;
import java.util.List;

import de.uniol.inf.is.odysseus.core.logicaloperator.LogicalOperatorCategory;
import de.uniol.inf.is.odysseus.core.predicate.IPredicate;
import de.uniol.inf.is.odysseus.core.sdf.schema.SDFSchema;
import de.uniol.inf.is.odysseus.core.server.logicaloperator.AbstractLogicalOperator;
import de.uniol.inf.is.odysseus.core.server.logicaloperator.UnaryLogicalOp;
import de.uniol.inf.is.odysseus.core.server.logicaloperator.annotations.LogicalOperator;
import de.uniol.inf.is.odysseus.core.server.logicaloperator.annotations.Parameter;
import de.uniol.inf.is.odysseus.core.server.logicaloperator.builder.BooleanParameter;
import de.uniol.inf.is.odysseus.core.server.logicaloperator.builder.PredicateParameter;
import de.uniol.inf.is.odysseus.core.server.physicaloperator.IHasPredicates;

@LogicalOperator(name = "ROUTE", minInputPorts = 1, maxInputPorts = 1, doc = "This operator can be used to
route the elements in the stream to different further processing operators, depending on the predicate.",
category = { LogicalOperatorCategory.PROCESSING })
public class RouteAO extends UnaryLogicalOp implements IHasPredicates{

    private static final long serialVersionUID = -8015847502104587689L;

    private boolean overlappingPredicates = false;
    private List<IPredicate<?>> predicates = new LinkedList<IPredicate<?>>();

    /**
     * if an element is routed to an output, heartbeats will be send to all
     * other outputs.
     */
    private boolean sendingHeartbeats = false;

    public RouteAO() {
        super();
    }

    public RouteAO(RouteAO routeAO) {
        super(routeAO);
        this.overlappingPredicates = routeAO.overlappingPredicates;
        this.sendingHeartbeats = routeAO.sendingHeartbeats;
        if (routeAO.predicates != null) {
            for (IPredicate<?> pred : routeAO.predicates) {
                this.predicates.add(pred.clone());
            }
        }
    }

    @Override
    protected SDFSchema getOutputSchemaIntern(int pos) {
        // since it is a routing, schema is always from input port 0
        return super.getOutputSchemaIntern(0);
    }

    @Parameter(type = PredicateParameter.class, isList = true)
    public void setPredicates(List<IPredicate<?>> predicates) {
        this.predicates = predicates;
    }

    @Override
    public List<IPredicate<?>> getPredicates() {
        return predicates;
```

```
    }

    @Parameter(name = "overlappingPredicates", type = BooleanParameter.class, optional = true, doc = "Evaluate
all (true) or only until first true predicate (false), i.e. deliver to all ports where predicate is true or
only to first")
    public void setOverlappingPredicates(boolean overlappingPredicates) {
        this.overlappingPredicates = overlappingPredicates;
    }

    @Override
    public AbstractLogicalOperator clone() {
        return new RouteAO(this);
    }

    /**
     * @return
     */
    public boolean isOverlappingPredicates() {
        return this.overlappingPredicates;
    }

    @Parameter(name = "sendingHeartbeats", type = BooleanParameter.class, optional = true, doc = "If an element
is routed to an output, heartbeats will be send to all other outputs")
    public void setSendingHeartbeats(boolean sendingHeartbeats) {
        this.sendingHeartbeats = sendingHeartbeats;
    }

    public boolean isSendingHeartbeats() {
        return this.sendingHeartbeats;
    }

}
```

Every operator needs to provide an output schema, i.e. what is the schema of the elements that are produced. As a default implementation *AbstractLogical Operator* delivers the input schema as the output schema. Route does not change the input schema, so this implementation is sufficient**. If the input schema is not the same as the output schema, the class needs to implement *getOutputSchemaIntern(int port)*, where port is the output port of the operator.**

Step 1b: Annotating the logical operator

For the easy integration of new logical operators into to PQL query language, annotations should be used. Another way is to extend *AbstractOperatorBuild er*. In these annotations the name of the operator and number of minimal and maximal inputs and the parameters are described. In addition, the annotations may include some documentation and a URL for further information about the operator. These information are also available in the Odysseus Studio GUI.
On the right side, the whole implementation of the RouteAO can be found. Because predicates need to be initialized before the processing, they should be saved by the *AbstractLogicalOperator*.
Further information about the annotations can be found in the PQL documentation.

Changing the output schema

For many operators the input schema and the output schema are the same but often you need to create you own output schema. For this, overwrite the method getInputSchemaInternal(port).

If you just want to merge the input from the left and the right, you just can use the method as in the following (from EnrichAO)

```
    @Override
    public synchronized SDFSchema getOutputSchemaIntern(int pos) {
        // The Sum of all InputSchema
        Iterator<LogicalSubscription> iter = getSubscribedToSource().iterator();
        SDFSchema left = iter.next().getSchema();
        SDFSchema right = iter.next().getSchema();
        SDFSchema outputSchema = SDFSchema.join(left,right);

        setOutputSchema(outputSchema);
        return outputSchema;
    }
```

If you want to add new Attributes, it is important that you create a new schema based on the input one (because there are many hidden stream descriptions inside the schema). In the following there is an example for this

```
    @Override
    public SDFSchema getOutputSchemaIntern(int pos) {

        final String start = "meta_valid_start";
        final String end = "meta_valid_end";

                // Create new Attributes
        SDFAttribute starttimeStamp = new SDFAttribute(null, start,
                SDFDatatype.TIMESTAMP, null, null, null);
        SDFAttribute endtimeStamp = new SDFAttribute(null, end,
                SDFDatatype.TIMESTAMP, null, null, null);

        List<SDFAttribute> outputAttributes = new ArrayList<SDFAttribute>();
            // Retrieve old attributes (they should all be part of the output schema)
        outputAttributes.addAll(getInputSchema(0).getAttributes());

            // add new Attributes
        outputAttributes.add(starttimeStamp);
        outputAttributes.add(endtimeStamp);

                // Create new Schema with Factory, keep input Schema!
        SDFSchema schema = SDFSchemaFactory.createNewWithAttributes(outputAttributes, getInputSchema(0)));
        return schema;
    }
```

You can also create a totally new schema, e.g. if your operator creates a different output from the input. Here is an example for this:

```
@Override
public SDFSchema getOutputSchemaIntern(int pos) {
        SDFAttribute out1 = new SDFAttribute(...);
        SDFAttribute out2 = new SDFAttribute(...);
        SDFAttribute out3 = new SDFAttribute(...);

        List<SDFAttribute> outAttributes = new ArrayList<>();
        outAttributes.add(out1);
        outAttributes.add(out2);
        outAttributes.add(out3);

        // Create new Schema with Factory, keep input Schema!
    SDFSchema schema = SDFSchemaFactory.createNewWithAttributes(outputAttributes, getInputSchema(0)));
    return schema;
}
```

Step 2: Create the physical operator

Until now, only descriptive information about the new operator is given. Next the concrete implementation of its functionality needs to be provided.
To create an own physical operator, *AbstractPipe<R,W>* needs to be extended. To allow generic implementations we utilize the Java generics approach. *R* is the type of the objects that are read by the operator and *W* is the type of the objects that are written. Although in most cases, this will be the same class (e.g. Tuple) some operators may read other objects than they write.
The following methods need to be overwritten:

- *getOutputMode()*: Needed for locking, Goal: Reduce object copies
    - INPUT: read element will not be modified (e.g. selection)
    - MODIFIED_INPUT: read element will be modified (e.g. projection)
    - NEW_ELEMENT: operator creates a new element (e.g. join)

```
    @Override
    public OutputMode getOutputMode() {
        return OutputMode.INPUT;
    }
```

Important: For operators with more than one input the processing must be synchronized. Especially, process_next and processPunctuation!

- *process_next(R input, int port)*
    - Process new input element on input port, a new created element can be send to the next operator with the *transfer()* method

```
@Override
protected void process_next(T object, int port) {
    for (int i=0;i<predicates.size();i++){
        if (predicates.get(i).evaluate(object)) {
            transfer(object,i);
            return;
        }
    }
    transfer(object,predicates.size());
}
```

If necessary: process_open, process_close. These methods are called when the query is initialized and terminated, respectively.

- *processPunctuation(Punctuation punctuation, int port)*
    - This method is needed to process punctuations (e.g.heartbeat, Heartbeats allow determining the progress of time without the need to produce new elements). It is important, that punctuations and elements are timely ordered, i.e. if a punctuation is send, no other element is allowed to be send with an older timestamp anymore! If the operator has no state, it is enough to call *sendPunctuation*.

```
@Override
public void processPunctuation(PointInTime timestamp, int port) {
    sendPunctuation(timestamp);
}
```

- *process_isSemanticallyEqual(IPhysicalOperator ipo)*
    - This method is needed for query sharing. Return true, if the operator given and the current operator are equivalent and can be replaced by each other.

```
@Override
public boolean process_isSemanticallyEqual(IPhysicalOperator ipo) {
    if(!(ipo instanceof RoutePO)) {
        return false;
    }
    RoutePO spo = (RoutePO) ipo;
    if(this.hasSameSources(spo) &&
            this.predicates.size() == spo.predicates.size()) {
        for(int i = 0; i<this.predicates.size(); i++) {
            if(!this.predicates.get(i).equals(spo.predicates.get(i))) {
                return false;
            }
        }
        return true;
    }
    return false;
}
```

As a naming convention the class name should end with *PO* (physical operator).

RoutePO

```
package de.uniol.inf.is.odysseus.tutorial.physicaloperator;

/********************************************************************************
 * Copyright 2011 The Odysseus Team
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
```

```
 *
 *     http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */

import java.util.ArrayList;
import java.util.Collection;
import java.util.List;

import de.uniol.inf.is.odysseus.core.metadata.IMetaAttribute;
import de.uniol.inf.is.odysseus.core.metadata.IStreamObject;
import de.uniol.inf.is.odysseus.core.metadata.ITimeInterval;
import de.uniol.inf.is.odysseus.core.physicaloperator.Heartbeat;
import de.uniol.inf.is.odysseus.core.physicaloperator.IPhysicalOperator;
import de.uniol.inf.is.odysseus.core.physicaloperator.IPunctuation;
import de.uniol.inf.is.odysseus.core.predicate.IPredicate;
import de.uniol.inf.is.odysseus.core.server.physicaloperator.AbstractPipe;
import de.uniol.inf.is.odysseus.core.server.physicaloperator.IHasPredicates;

/**
 * @author Marco Grawunder
 */
@SuppressWarnings({ "rawtypes" })
public class RoutePO<T extends IStreamObject<IMetaAttribute>> extends AbstractPipe<T, T> implements
IHasPredicates {

        private List<IPredicate<?>> predicates;
        final boolean overlappingPredicates;

        /**
         * if an element is routed to an output, heartbeats will be send to all other outputs.
         */
        final boolean sendingHeartbeats;

        public RoutePO(List<IPredicate<T>> predicates,
                        boolean overlappingPredicates, boolean sendingHeartbeats) {
                super();
                this.overlappingPredicates = overlappingPredicates;
                this.sendingHeartbeats = sendingHeartbeats;
                initPredicates(predicates);
        }

        @Override
        public List<IPredicate<?>> getPredicates() {
                return predicates;
        }

        private void initPredicates(List<IPredicate<T>> predicates) {
                this.predicates = new ArrayList<IPredicate<?>>(
                                predicates.size());
                for (IPredicate<?> p : predicates) {
                        this.predicates.add(p.clone());
                }
        }

        @Override
        public OutputMode getOutputMode() {
                return OutputMode.INPUT;
        }

        @SuppressWarnings("unchecked")
        @Override
        protected void process_next(T object, int port) {
                boolean found = false;
                Collection<Integer> routedToPorts = null;
                if(sendingHeartbeats) {
```

```
                routedToPorts = new ArrayList<>();
            }
            for (int i = 0; i < predicates.size(); i++) {
                if (((IPredicate<T>)predicates.get(i)).evaluate(object)) {
                    T out = object;
                    // If object is send to multiple output ports
                    // it MUST be cloned!
                    if (overlappingPredicates) {
                        out = (T)object.clone();
                        out.setMetadata(object.getMetadata().clone());
                    }
                    transfer(out, i);
                    found = true;
                    if ((sendingHeartbeats) && (routedToPorts != null)) {
                        routedToPorts.add(i);
                    }
                    if (!overlappingPredicates) {
                        break;
                    }
                }
            }
            if (!found) {
                transfer(object, predicates.size());
                if ((sendingHeartbeats) && (routedToPorts != null)) {
                    routedToPorts.add(predicates.size());
                }
            }

            if ((sendingHeartbeats) && (routedToPorts != null)) {
                // Sending heartbeats to all other ports
                for(int i = 0; i < predicates.size(); i++) {

                    if(!routedToPorts.contains(i))
                        this.sendPunctuation(Heartbeat.createNewHeartbeat(((IStreamObject<?
extends ITimeInterval>) object).getMetadata().getStart()), i);

                }
            }
        }

        @Override
        public void processPunctuation(IPunctuation punctuation, int port) {
            for (int i=0;i<predicates.size();i++){
                sendPunctuation(punctuation,i);
            }
        }

        @Override
        public boolean process_isSemanticallyEqual(IPhysicalOperator ipo) {
            if (!(ipo instanceof RoutePO)) {
                return false;
            }
            RoutePO spo = (RoutePO) ipo;
            if (this.predicates.size() == spo.predicates.size()) {
                for (int i = 0; i < this.predicates.size(); i++) {
                    if (!this.predicates.get(i).equals(spo.predicates.get(i))) {
                        return false;
                    }
                }
                return true;
            }
            return false;
        }

        @Override
        public void setPredicates(List<IPredicate<?>> predicates) {
            this.predicates = predicates;
        }

}
```

TODO: InputSyncArea, OutputTransferArea

Important: If you create a new StreamObject at runtime you must copy the cloned metadata!

```
out.setMetadata(input.getMetadata().clone());
```

Query Rewrite (Step 3)

Rewriting is used to switch, add, remove or replace logical operators before they are transformed into their physical counterparts. Normally, the aim of the rewriting process is to optimize the query, for example, by pushing down selection operator before costly joins. The implementation is done by rewriting rules, which are implemented like transformation rules (see next section).

Transformation (Step 4)

To translate the logical operator into the physical counterpart the transformation engine is needed. The rule for the route operator can be found in the following example. Further information can be found in the description of the transformation component.

TRouteAO

```
/**********************************************************************************
  * Copyright 2011 The Odysseus Team
  *
  * Licensed under the Apache License, Version 2.0 (the "License");
  * you may not use this file except in compliance with the License.
  * You may obtain a copy of the License at
  *
  *     http://www.apache.org/licenses/LICENSE-2.0
  *
  * Unless required by applicable law or agreed to in writing, software
  * distributed under the License is distributed on an "AS IS" BASIS,
  * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
  * See the License for the specific language governing permissions and
  * limitations under the License.
  */
package de.uniol.inf.is.odysseus.tutorial.rules;

import de.uniol.inf.is.odysseus.core.server.logicaloperator.RouteAO;
import de.uniol.inf.is.odysseus.core.server.physicaloperator.RoutePO;
import de.uniol.inf.is.odysseus.core.server.planmanagement.TransformationConfiguration;
import de.uniol.inf.is.odysseus.ruleengine.rule.RuleException;
import de.uniol.inf.is.odysseus.ruleengine.ruleflow.IRuleFlowGroup;
import de.uniol.inf.is.odysseus.transform.flow.TransformRuleFlowGroup;
import de.uniol.inf.is.odysseus.transform.rule.AbstractTransformationRule;

@SuppressWarnings({"unchecked","rawtypes"})
public class TRouteAORule extends AbstractTransformationRule<RouteAO> {

        @Override
        public int getPriority() {
                return 0;
        }


        @Override
        public void execute(RouteAO routeAO, TransformationConfiguration config) throws RuleException
{
                defaultExecute(routeAO, new RoutePO(routeAO.getPredicates(), routeAO.isOverlappingPredicates(),
routeAO.isSendingHeartbeats()), config, true, true);
        }

        @Override
        public boolean isExecutable(RouteAO operator, TransformationConfiguration transformConfig) {
                return operator.isAllPhysicalInputSet();
        }

        @Override
        public String getName() {
                return "RouteAO -> RoutePO";
        }

        @Override
        public IRuleFlowGroup getRuleFlowGroup() {
                return TransformRuleFlowGroup.TRANSFORMATION;
        }

        @Override
        public Class<? super RouteAO> getConditionClass() {
                return RouteAO.class;
        }

}
```