# optimizer components - from scratch

Documentation: https://support.quintiq.com/doc/libopt

Library:
https://solutions.quintiq.com/solutions/systemstartpage?solution=Optimization&system=Optimizer+Components

## Build your model

We assume you have a Quill model of your puzzle: orders, paper rolls, routes, resources, etc. For this document, we will assume there are objects Order, PaperRoll, OrderOnPaperRoll with obvious meanings.

## Add the LibOpt library to your project

- Follow the instructions from the Components Library documentation on adding the library to your project and linking the module to your model.
- Keep following those instructions to create an object of base type `LibOpt_Optimization` and instantiate it from your dataset. In the e-learning, this object is called `OptimizationQuiCo.` It is owned by `Company` and instantiated in the On Constructed… tab of the dataset.
- Create the `LibOpt_Optimizer` subclass "`OptimizerQuiCo`". Note that we have two abstract methods that need to be overridden.
- Override `OptimizationQuiCo::UpdateOptimizers()` and enter this code:

  ```
              this.UpdateOptimizer( typeof( OptimizerQuiCo ) );
  ```

  This registers OptimizerQuiCo in the OptimizationQuiCo object, and makes sure an instance is available in the "Optimizers" form when the model is run.
- Select `OptimizerQuiCo`, and override the method `CreateComponents`. This is the entry point where we will build our optimizer, that is, where the component layout is defined.
- Add the GUI elements through Designer. Again, just follow the instructions from the Components Library documentation.

Note: your model should have only one Optimization object, but can have several Optimizer objects, one for each separate optimization algorithm.

## Create scope elements

Each puzzle element (*order* and *paper roll* in this case, but more generally it can include *operation, resource, supply, routing, shipment, …*) can be part of some subpuzzles but not others. The Components library has a unified way of defining this using *scope elements*. Each object that corresponds to a puzzle element gets its own subclass of `LibOpt_ScopeElement`. Subsets of these are collected in *scopes*.

- Create two subclasses of `LibOpt_ScopeElement`. Name them `ScopeElementPaperRoll` and `ScopeElementOrder`.
- Create 1-1 owning relations from `PaperRoll` and from `Order` (so `PaperRoll` owns `ScopeElementPaperRoll` and `Order` owns `ScopeElementOrder`).
- Each scope element has an *identifier*, which you will see back in debugging sessions. Override the `CalcIdentifier` method in each, and compute a meaningful value. For ScopeElementPaperRoll this could look as follows:

```
value := 'paper:' + [String]this.PaperRoll().MaterialNr();
this.Identifier( value );
```

- To make sure the scope elements are created and made available to the optimizer, override the method `OptimizerQuiCo::DefaultScope()` and enter the following (easy to understand) code for its body:

```
result := construct( LibOpt_ScopeElements );
company := this.Optimization().astype( OptimizationQuiCo ).Company();

traverse( company, Order, order )
{
  elem := order.ScopeElementOrder();
  if( isnull( elem ) )
  {
    elem := order.ScopeElementOrder( relnew );
  }
  result.Add( elem );
}
traverse( company, PaperRoll, paperroll )
{
  elem := paperroll.ScopeElementPaperRoll();
  if( isnull( elem ) )
  {
    elem := paperroll.ScopeElementPaperRoll( relnew );
  }
  result.Add( elem );
}

return & result;
```

### Accessing scope elements

Since a scope is essentially just a subset of scope elements, we would like ways to access the scope elements of a particular type. For example, a traverse of all orders in a scope might look like this:

```
traverse( scope.ScopeElements(),
          Elements.astype( ScopeElementOrder ).Order,
          order )
```

Since we will do this regularly, we want to create shortcuts. Overriding the LibOpt_Scope object for this purpose seems overkill, so we will *extend* the library object with two new methods.

- Add a method `Orders()` to the `LibOpt_Scope` type, with return type `Orders`, that returns all `Order` instances whose `ScopeElementOrder` is in the scope. The method should be "return owning" and have the following body:

```
value := selectset( this.ScopeElements(),
                    Elements.astype( ScopeElementOrder ).Order,
                    order, true );
return & value;
```

- Add a similar method `PaperRolls().`

**At this point, you have reached the starting model of the e-learning.**