MathModelica® System Designer™

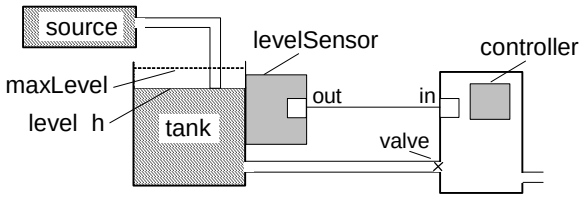
Tank System

1 Introduction

This example illustrates how you can build a hierarchical model using MathModelica System Designer. First a flat tank model is developed, then a similar component based is model, and at the end we show the flexibility that this gives us to test new scenarios. The example illustrates how to make new libraries.

2 Flat tank

The system we will begin with is a one tank system with controller as illustrated in the picture below.



To implement the model we need to setup the system equations. The water level, *h*, in the tank is a function of the flow in and out from the tank, and the tank area:

$$\dot{h} = \frac{q_{in} - q_{out}}{A}$$

For this example we will choose an input flow that is constant the first 150 seconds and then triples after this

$$q_{in} = \begin{cases} flowLevel & t < 150s \\ 3 * flowLevel & t \geq 150s \end{cases}$$

where *flowLevel* is a parameter. By controlling the output flow we will try to hold the tank level at a desired reference value, *ref*. For this goal we implement a PI controller:

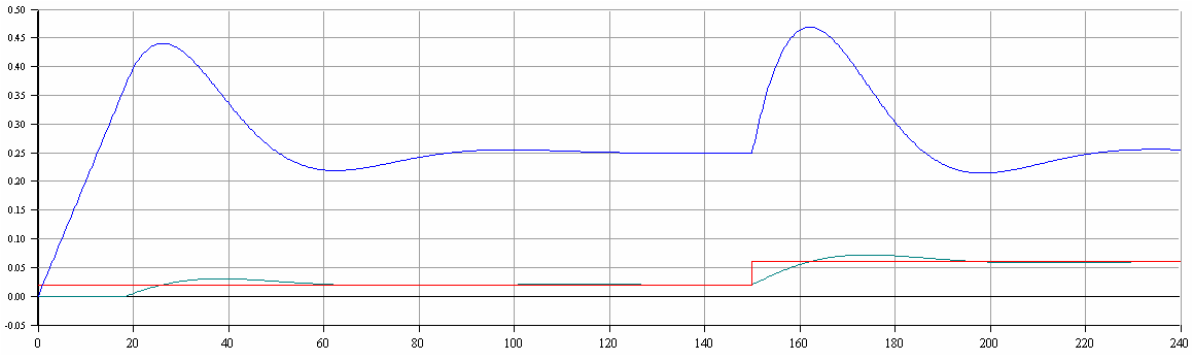
$$q_{out} = K\left(error(t) + \frac{1}{T}\int_0^t error(s)ds \right)$$

where *K* is the controller gain and *T* is the time constant of the controller. Finally we limit the output flow to be within a minimum value, *minV,* and a maximum, *maxV*. With this information we can now implement the flat Modelica code.

MathCore Engineering AB
Teknikringen 1B
SE-583 30 Linköping, Sweden
Phone +46 13 32 85 00
Fax +46 13 21 27 01
E-mail info@mathcore.com
Web: www.mathcore.com

```
model FlatTank
  parameter Real flowLevel(unit="m3/s")=0.02;
  parameter Real area(unit="m2")=1;
  parameter Real flowGain(unit="m2/s")=0.05;
  parameter Real K=2 "Gain";
  parameter Real T(unit="s")=10 "Time constant";
  parameter Real minV=0,maxV=10;
  parameter Real ref=0.25 "Reference level for control";
  Real h(start=0,unit="m") "Tank level";
  Real qInflow(unit="m3/s") "Flow through input valve";
  Real qOutflow(unit="m3/s") "Flow through output valve";
  Real error "Deviation from reference level";
  Real outCtr "Control signal without limiter";
  Real x "State variable for controller";

equation
  assert(minV >= 0, "minV must be greater or equal to zero");
  der(h)=(qInflow - qOutflow)/area;
  qInflow=if time > 150 then 3*flowLevel else flowLevel;
  qOutflow=Functions.LimitValue(minV, maxV, -flowGain*outCtr);
  error=ref - h;
  der(x)=error/T;
  outCtr=K*(error + x);
end FlatTank;

function LimitValue
  input Real pMin;
  input Real pMax;
  input Real p;
  output Real pLim;
algorithm
  pLim:=if p > pMax then pMax else if p < pMin then pMin else p;
end LimitValue;
```

By simulating the model for 250 seconds we can see that the tank level starts to increase reaching and passing the desired reference level. When the desired level is passed the outflow is opened and after 150 seconds the level has stabilized. However at this moment the input flow is suddenly increased and the water level is therefore increased before the controller manages to stabilize it again. This is illustrated in the figure below.

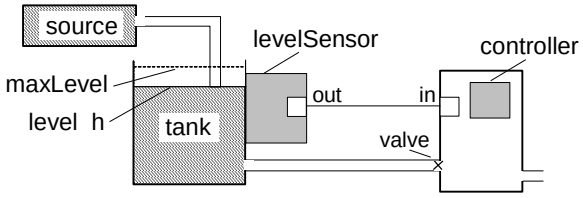


3 Component Based Tank

Implementing a component based tank will require a little bit more work to begin with, but as soon as we want to start experimenting with the tank and test different scenarios we will regain the invested time.

When using the object-oriented component-based approach to modeling we first try to understand the system structure and decomposition in a hierarchical top-down manner. When the system components and interactions between these components have been roughly identified, we can apply the first traditional modeling phases of identifying variables and equations on each of these model components.

Going back to the original drawing we can see that the tank system has a natural component structure, as illustrated below.



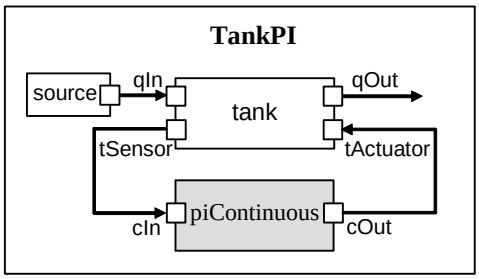
We can identify five components in the figure: the tank itself, the liquid source, the level sensor, the valve, and the controller. However, since we will choose very simple representations of the level sensor and the valve, i.e. just a simple scalar variable for each of these two components, we let these variables be simple Real variables in the tank model instead of creating two new classes containing a single variable each.

The next step is to determine the interactions and communication paths between the components. It is fairly obvious that fluid flows from the source tank via a pipe. Fluid also leaves the tank via an outlet controlled by the valve. The controller needs measurements of the fluid level from the sensor. Thus a communication path from the sensor of the tank and the controller needs to be established.

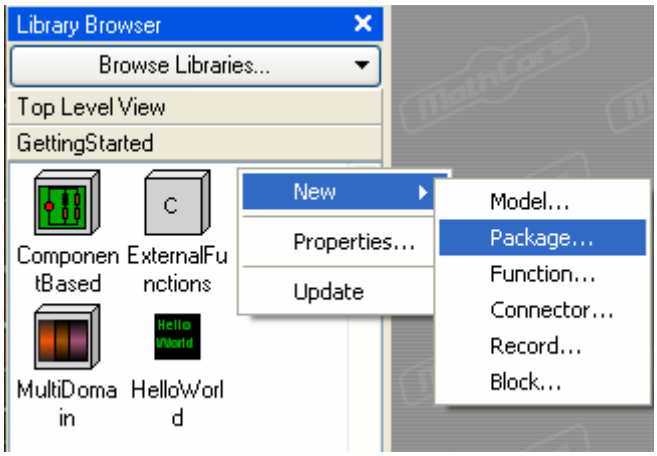
Communication paths need to be connected somewhere. Therefore, connector instances need to be created for those components which are connected, and connector classes declared when needed. In fact, the system model should be designed such that the only communication between a component and the rest of the system is via connectors.

Finally we should think about reuse and generalizations of certain components. Do we expect several variants of a component to be needed? In the case of the tank system we expect to plug in several variants of the controller, starting with a PI controller. Thus it is useful for us to create a base class for tank system controllers.

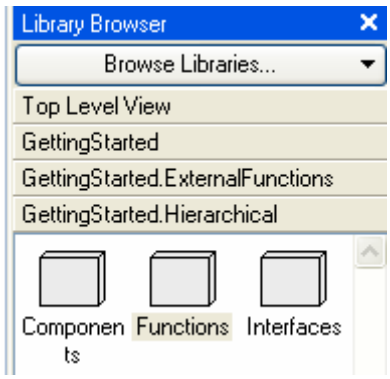
The structure of the tank system model developed using the object-oriented component-based approach is clearly visible in the figure below:



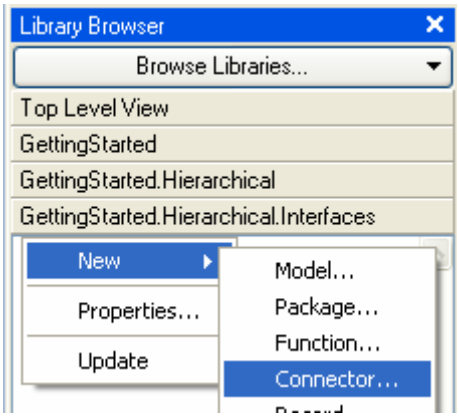
We can identify three different types of classes that will be used in the model: interfaces, functions and components. Therefore we develop a package containing three sub packages. To create a package within the GettingStarted package we right-click in the library browser and select package:



In the dialog box that opens we choose a name, Hierarchical, and click ok. Then we open this package and create Components, Functions, and Interfaces packages within this package:



After this we are ready to create the interfaces, called connectors. This is done by opening the Interfaces package and the creating connector classes within it:



We create one connector for reading the fluid level (see the Hello world example to see how to develop model textually and make model icons):

```
connector ReadSignal "Reading fluid level"
   Real val(unit="m");
end ReadSignal;
```

One for the signal to actuator for setting valve position:

```
connector ActSignal "Signal to actuator for setting valve position"
   Real act;
end ActSignal;
```

And on for the liquid flow at inlets and outlets:

```
connector LiquidFlow "Liquid flow at inlets or outlets"
   Real lflow(unit="m3/s");
end LiquidFlow;
```

Next step is to create the three components of the system. These are created in the Components package. We begin with the tank, which has four connectors: qIn for input flow, qOut for output flow, tSensor for providing fluid level measurements, and tActuator for setting the position of the valve at the outlet of the tank. The central equation regulating the behavior of the tank is the *mass balance* equation, which in the current simple form assumes constant pressure. The output flows are related to the valve position by a flowGain parameter, and by the limiter that guarantees that the flow does not exceed what corresponds to the open/closed positions of the valve:

```
model Tank;
   parameter Real area(unit="m2")=0.5;
   parameter Real flowGain(unit="m2/s")=0.05;
   parameter Real minV=0,maxV=10;
   Real h(start=0.0,unit="m") "Tank level";
   GettingStarted.Hierarchical.Interfaces.ReadSignal tSensor "Connector,
sensor reading tank level (m)";
   GettingStarted.Hierarchical.Interfaces.LiquidFlow qIn "Connector, flow
(m3/s) through input valve";
   GettingStarted.Hierarchical.Interfaces.LiquidFlow qOut "Connector, flow
(m3/s) through output valve";
   GettingStarted.Hierarchical.Interfaces.ActSignal tActuator "Connector,
actuator controlling input flow";

equation
assert(minV >= 0, "minV ? minimum Valve level must be >= 0 ");
  der(h)=(qIn.lflow - qOut.lflow)/area;

equation
  qOut.lflow=Functions.LimitValue(minV, maxV, -flowGain*tActuator.act);
  tSensor.val=h;
end Tank;
```

The model uses the already defined connector, but also the LimitFunction, which has not been defined yet. This is defined by creating the following function in the Functions package:

```
function LimitValue;
  input Real pMin;
  input Real pMax;
  input Real p;
  output Real pLim;
algorithm
  pLim:=if p > pMax then pMax else if p < pMin then pMin else p;
end LimitValue;
```

The fluid entering the tank must come from somewhere. Therefore we have a liquid source component in the tank system Flow increases sharply at t=150 to factor of three of the previous flow level, which creates an interesting control problem that the controller of the tank has to handle. The following model is created in the Components package:

```
model LiquidSource;
```

```
    parameter Real flowLevel=0.02;
    GettingStarted.Hierarchical.Interfaces.LiquidFlow qOut;

  equation
    qOut.lflow=if time > 150 then 3*flowLevel else flowLevel;
  end LiquidSource;
```

Finally the controllers need to be specified. We will initially choose a PI controller but later replace it by other kinds of controllers. The behavior of a PI (*proportional and integrating*) controller is primarily defined by the following two equations:

$$\frac{dx}{dt} = \frac{error}{T}$$

$$outCtr = K * (error + x)$$

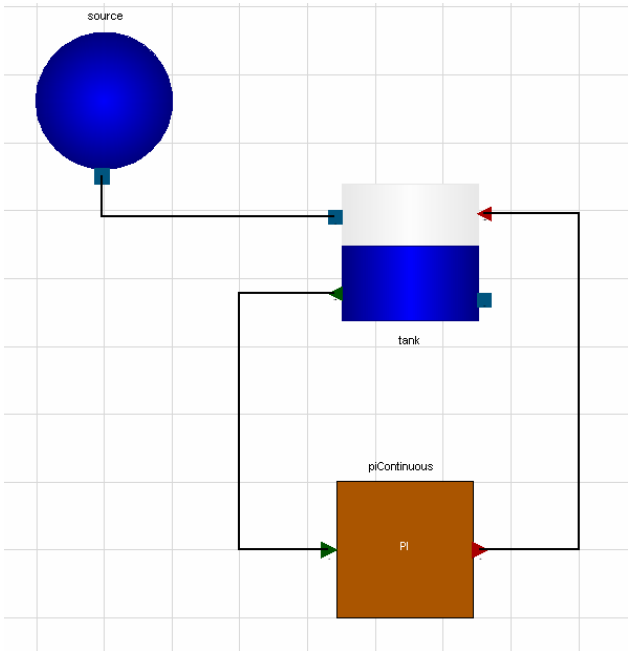
Here x is the controller state variable, *error* is the difference between the reference level and the actual level of liquid obtained from the sensor, T is the time constant of the controller, *outCtr* is the control signal to the actuator for controlling the valve position, and K is the gain factor. These two equations are placed in the controller class PIcontinuousController, which extends the BaseController class defined later:

```
   model PIcontinuousController
    extends BaseController(K=2,T=10);
    Real x "State variable of continuous PI controller";

  equation
    der(x)=error/T;
    outCtr=K*(error + x);
  end PIcontinuousController;
```

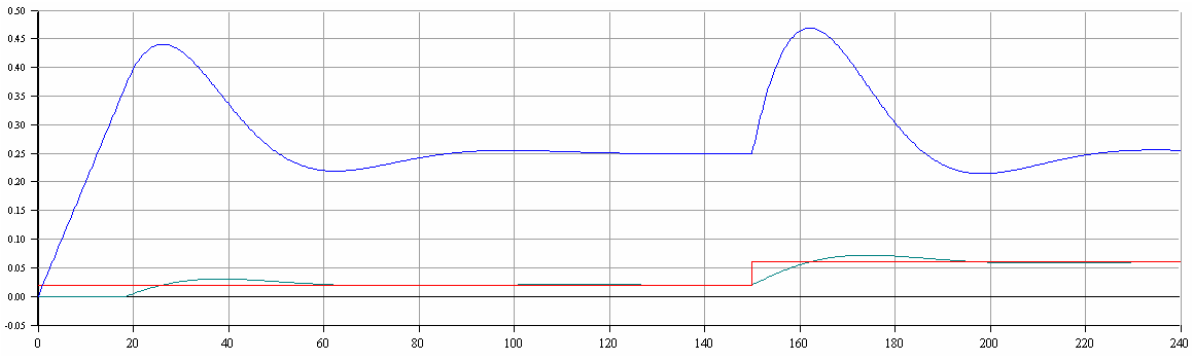
Both the PI and PID controller to be defined later inherit the partial controller class BaseController, containing common parameters, state variables, and two connectors: one to read the sensor and one to control the valve actuator.

```
  partial model BaseController;
    parameter Real Ts(unit="s")=0.1 "Time period between discrete samples";
    parameter Real K=2 "Gain";
    parameter Real T(unit="s")=10 "Time constant";
    parameter Real ref "Reference level";
    Real error "Deviation from reference level";
    Real outCtr "Output control signal";
    GettingStarted.Hierarchical.Interfaces.ReadSignal cIn "Input sensor
  level,  connector";
    GettingStarted.Hierarchical.Interfaces.ActSignal cOut "Control to
  actuator, connector";

  equation
    error=ref - cIn.val;
    cOut.act=outCtr;
  end BaseController;
```

When this is finished we can compose our model by drag and drop:



Simulating for 250 seconds yields the same result as the flat tank system:



4 Tank with continuous PID controller

We define a TankPID system which is the same as the TankPI system except that the PI controller has been replaced by a PID controller. Here we see a clear advantage of the object-oriented component-based approach over the traditional model-based approach, since system components can easily be replaced and changed in a plug-and-play manner.

A PID (*proportional*, *integrating*, *derivative*) controller model can be derived in a similar way as done for the PI controller. The basic equations for a PID controller are the following:

$$\frac{dx}{dt} = \frac{error}{T}$$

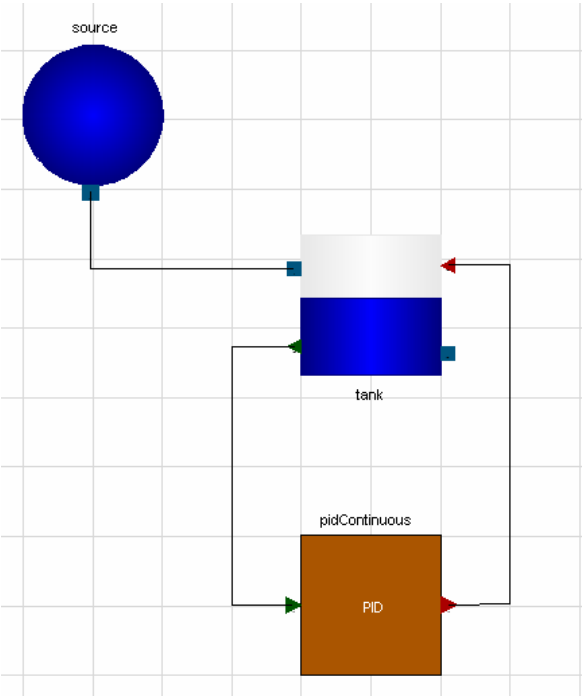
$$y = T\frac{derror}{dt}$$

$$outCtr = K * (error + x + y)$$

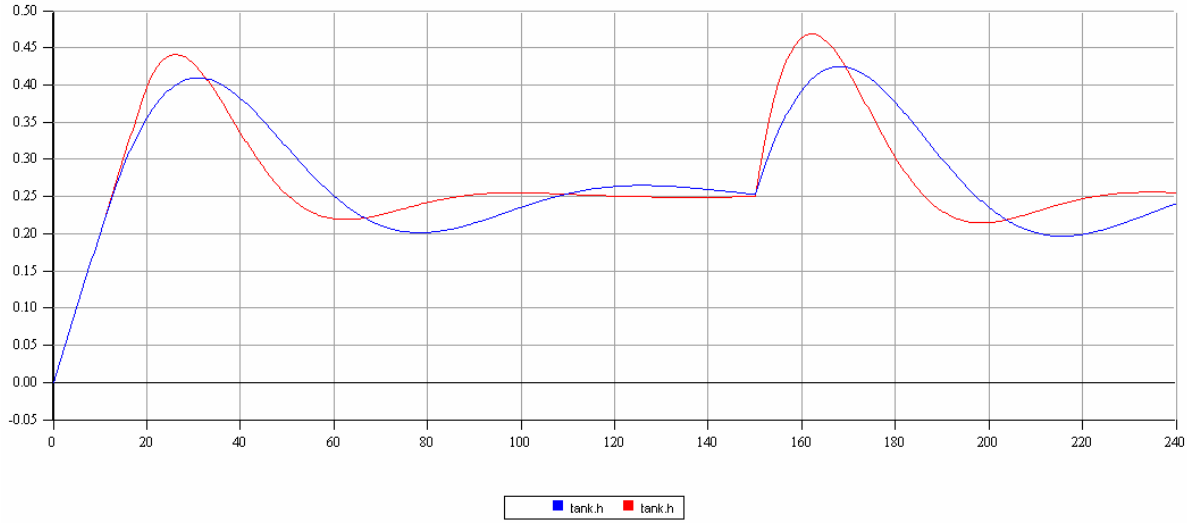
Using this equations and the BaseController class we create the PID controller:

```
model PIDcontinuousController
  extends BaseController(K=2,T=10);
  Real x "State variable of continuous PID controller";
  Real x "State variable of continuous PID controller";

equation
  der(x)=error/T;
  y=T*der(error);
  outCtr=K*(error + x + y);
end PIDcontinuousController;
```

We can now compose a PID controlled tank system using drag and drop:

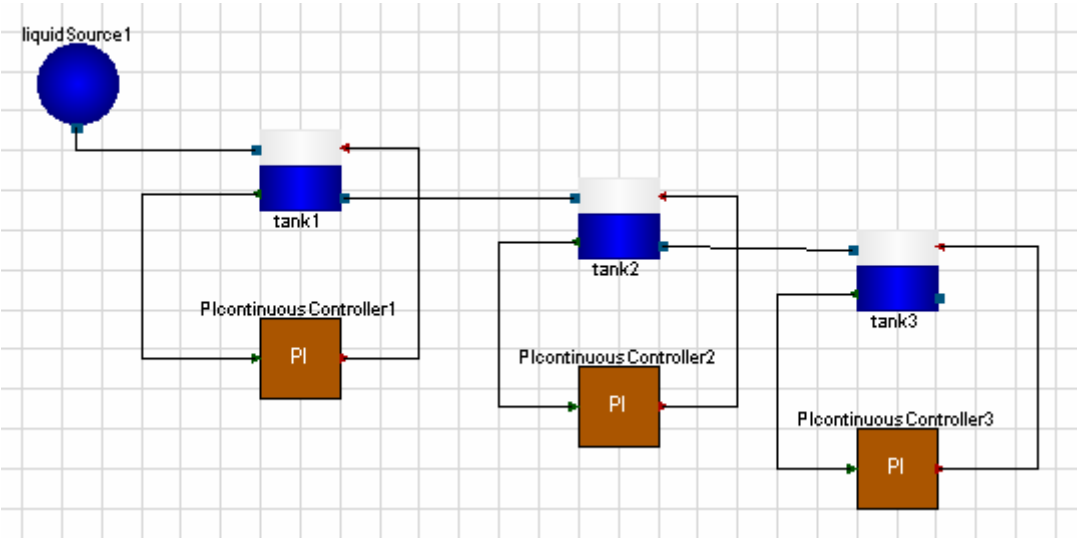


We simulate for 250 seconds again and compare with the previous result:

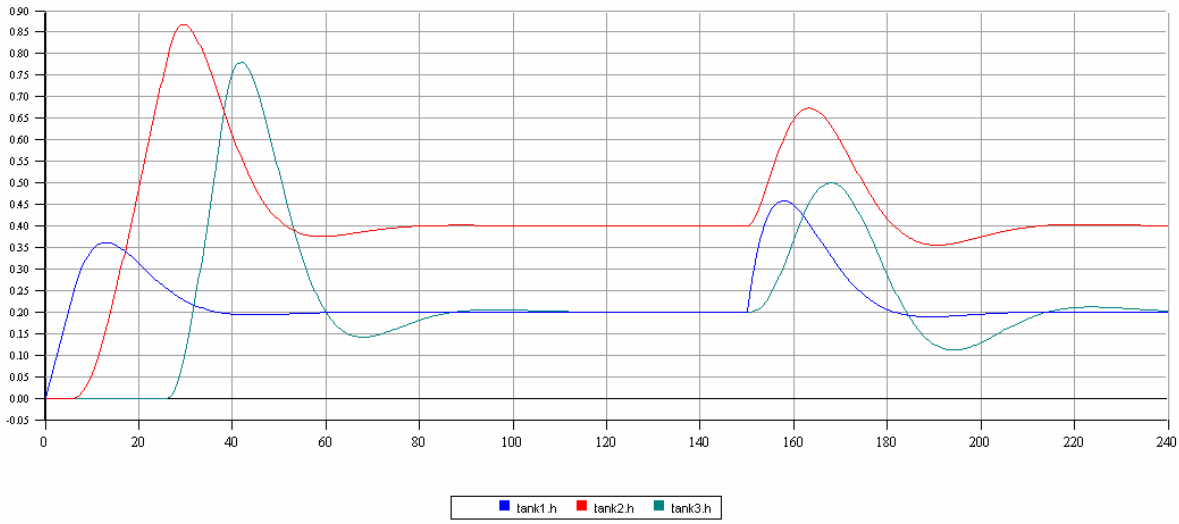


5 Tank System

Finally, thanks to the object-oriented component-based approach we can compose a larger system with ease:



Simulating this system we can now study how the tank level of each tank is controlled:



Note that the second tank has a reference level of 0.4 meters while the other tanks only have 0.2 meters.