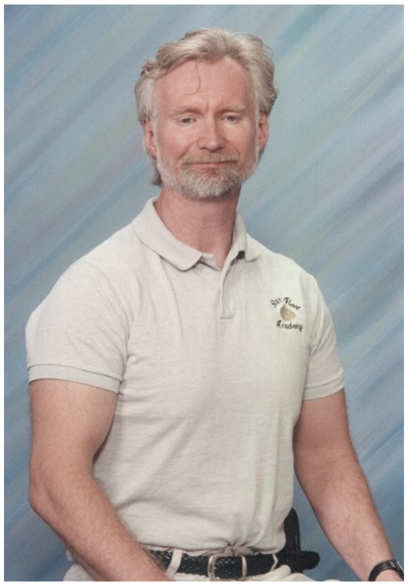
Any Port in a Storm



Bruce Powel Douglass
Chief Evangelist
I-Logix

Architectures in the UML 2.0

The UML 2.0 has a primary theme of "scalability." This shows up in many of the new and elaborated aspects of the UML, such as decomposable sequence diagrams (a major feature in the Rhapsody 5.0), structured classes (which have been in Rhapsody since the 1.0 days) and an improved mechanisms for enforcing collaboration – ports.

Ports aren't really a new concept. They have been around in systems engineering for a long time and ports have been a part of other design languages, although in a more limited form. Ports, as we shall see later, are more of a design patter[1]n than a fundamental design concept. As such, they have both pros and cons. They do facilitate the strong encapsulation of the internals of structured classes, but at the cost of increased design and code complexity. If the need for encapsulation is greater than the cost they incur, then ports would be an applicable design pattern – otherwise, other mechanisms for encapsulation should be used instead.

While structured classes and ports, as specified in the UML 2.0 have definite uses, particularly in specifying architectures, they have a number of subtle aspects. A goodly portion of the upcoming *Real-Time UML 3rd Edition: Advances in the UML for Real-Time Systems* (to be released in February, 2004) deal with the use of the structuring and encapsulating mechanisms of the new UML revision. Structurally, it all centers around the notion of a structured class, so let's begin there.

The Structured Class

A structured class is a class that contains parts, which are themselves typed by classes. Notice that I didn't say that a structure class contains classes, as this isn't strictly speaking true. The structured class does aggregate, via composition, classes, but these classes are the specifications of the parts, rather than actually being the parts. Just to confuse things, the parts aren't really objects either, even though at run-time, the part roles will be played by actual instances (objects). More precisely, the structured class

[1] See the ROOM Pattern in *Real-Time Design Patterns: Scalable Robust Architectures for Real-Time Systems* by Bruce Powel Douglass, PhD.

contains *contextualized object roles* (parts). *I know what you’re thinking* – “What does THAT mean???” Let me explain.

In vanilla UML 1.x, if we had an automobile that owned an engine that drove some wheels, we might draw something like Figure 1:

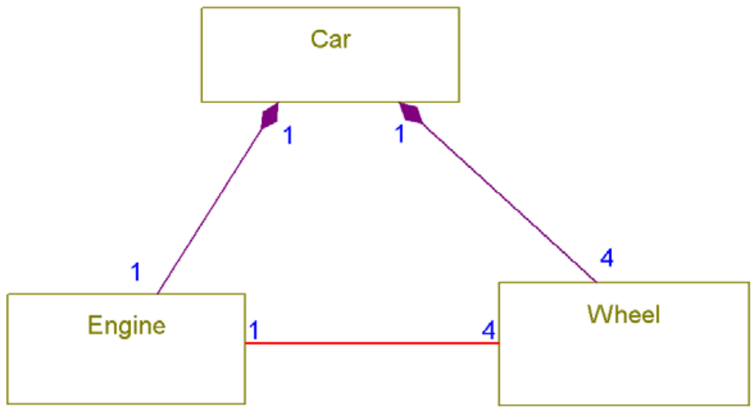


Figure 1: UML 1.x Car

While straightforward, this representation has a few problems. One of which is that while Engine associates with the Wheel class with a multiplicity of 4 (on the Wheel end), and the car strongly aggregates 4 Wheels, there is nothing that states the these are the SAME Wheels. This model could represent a system in which the Engine drives wheels on different cars! You can add constraints to the two relations to the Wheel class to enforce that, but you’d rather not do that because it clutters the diagram.

Another problem is shown in the next figure. In Figure 2, we want to reuse the Engine class in an entirely different context – we want it to drive a single propeller for a boat. In the UML 1.x, this is a problem, because the Engine class now has structurally two different associations – one to Propeller and one to Wheel.

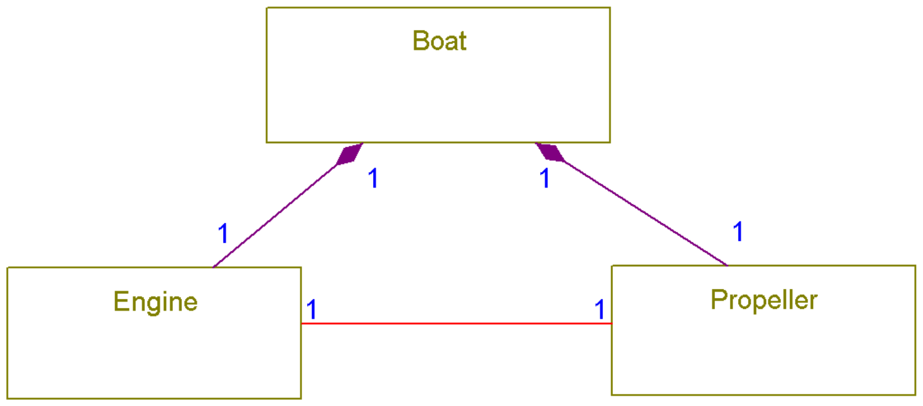


Figure 2: UML 1.x Boat

We can redraw the Car and Boat classes to address both of these concerns. In Figure 3, we see the both the Car and Boat classes drawn as structured classes. Inside of these classes, we see not classes, but *parts* – objects in a contextualized role. The notation for the name of the parts is the same as for an instance – "role name ';' class name". The role name is the name by which the structured class knows the part. The class name is the name of the class that types the part. "theEngine:Engine" has a part name of "theEngine" and that part is typed by class Engine.

So why is a part not an object, you say? An object is a specific instance of a class that exists at run-time, and occupies some specific memory location at some specific time. The parts of a structured class still specify the structure of the class, and they still exist at design-time, not run-time. When you draw a class with such parts, you're saying that all instances of this car class have these parts that make up their internal structure – so it is still a design-time specification of a car. During run-time, when I make an instance of the car, a specific instance of the Engine class (with some particular serial number) will be in that instance of car.

Also note that we don't show an association between the parts, rather, we show a *connection*. Just as a part is not a class, a connection is not an association – it is a contextualized link between parts in the context of a structured class that is typed by an association. At run-time, in some specific car, an actual link will fulfill the role specified by this connection. Interestingly, if you look closely at the browser view on the left-hand side of Figure 3, you will see that the connection between the Car parts (called the itsEngine itsWheel link) is actually owned by the Car and the connection between the Boat parts (called the itsEngine itsPropeller link) is owned by the Boat. This means that the structured class responsibility for wiring together its parts.

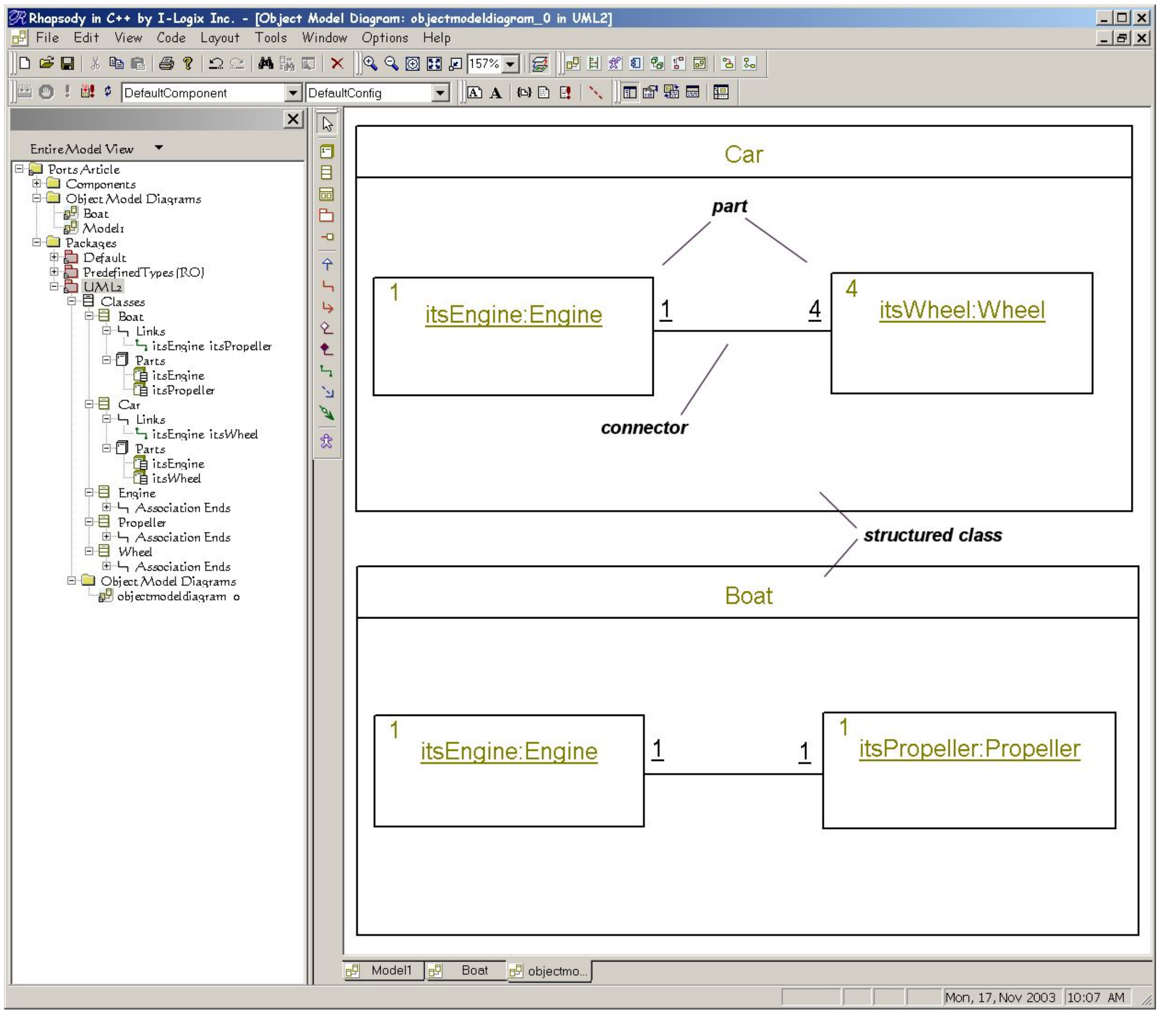


Figure 3: Structured Classes

The UML 2.0 draft specification defines a new specialized form of a class diagram called a *structure diagram* that basically subsets the normal UML class diagrams. Its purpose is to show the internal part structure of a single class. In Rhapsody 5.0, you can draw a structured class on a class diagram or on a structure diagram.

On to Ports

Let us suppose that a part of a structured class provides a service that we would like to invoke from a client of the structured class, something like the model shown in Figure 4. The Ventilator class has a itsO2Sensor part, typed by the O2Sensor class. This part offers a service getValue() which returns the measured value of the sensor. We would like the Display class to be able to retrieve that information and display it to the user. Suppose further, that we would like to make the eminently reasonable restriction that while the Display class “knows” that it should get the data from the Ventilator, that the Display class should know nothing about the internal structure of the Ventilator, including which part provides the service. How can we do that?

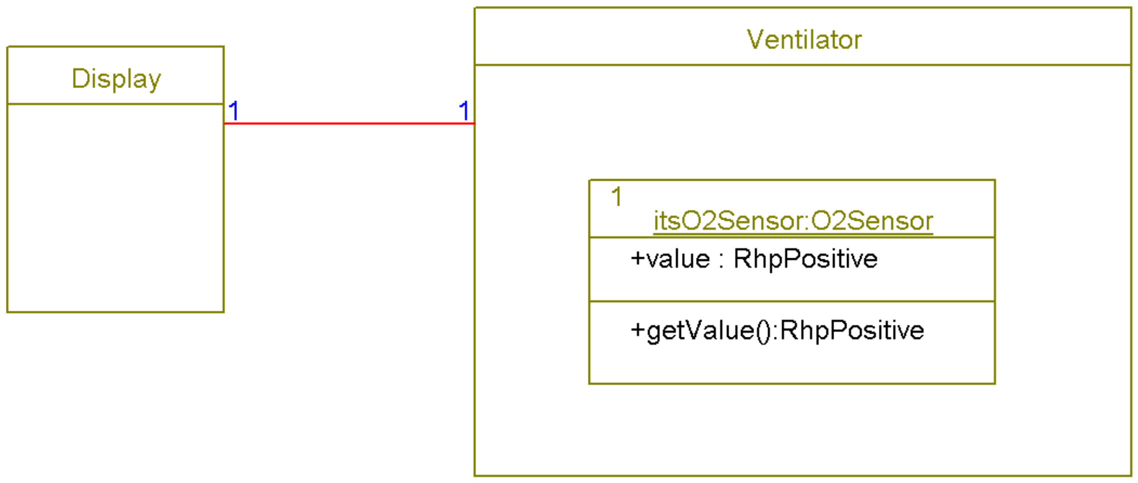


Figure 4: Clients of Structured Classes

One simple way is to provide an operation called getValue() on the Ventilator class. This operation would be invoked by the Display class and would return the value to that client. Internally, the method body for the getValue() would be something like

```
Ventilator::getValue() : RhpPositive {
      return itsO2Sensor.getValue();
}
```

That works, and is very flexible. If the value to be returned is to be collected from many different parts and then filtered before being returned, this is a flexible easy way to do just that. Furthermore, if we wanted to do it via an asynchronous rendezvous, we can pass events to request the information to the Ventilator and the Ventilator state machine would process that event with the line of code as it's action. However, how the return value is computed, which part(s) it comes from, and so on, is hidden down in the code or on an action list on a statechart. Wouldn't it be nice to have some way to indicate on the structural diagram that service is offered by the Ventilator, but ultimately provided by the internal itsO2Sensor part?

Ports do exactly that. The port version of Figure 4 is shown in Figure 5.

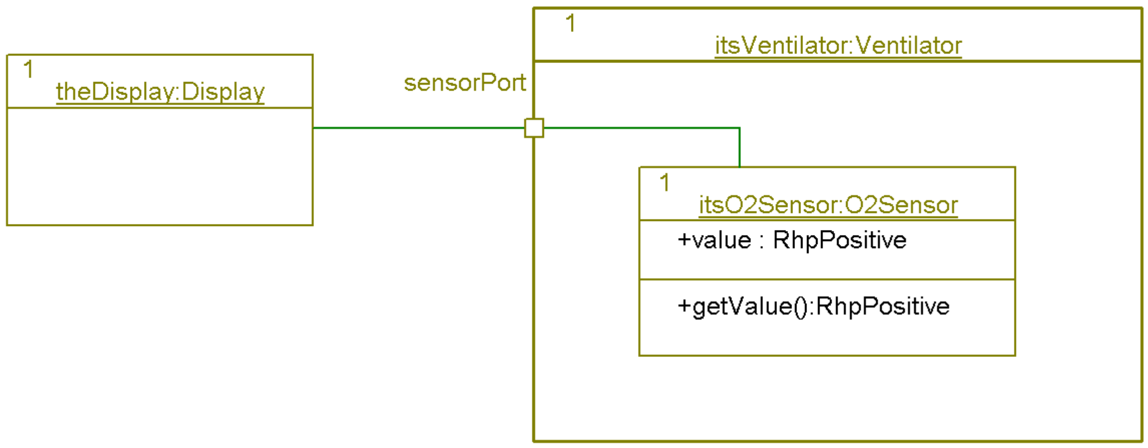


Figure 5: Clients and Ports

The port labeled "sensorPort" is on the boundary of the ventilator class. It links the Display class to the Ventilator class. When the service getValue() is invoked on the sensorPort, the sensorPort invokes the service on the internal itsO2Sensor part. Be sure to note that the connection between the Display class and the port is a link and not an association. The UML 2 spec doesn't allow for drawing associations to ports – a feature which I believe to be a flaw. Because of this requirement, we must use an instance of the Display class (called, in this case, theDisplay) to which to link.

Ports are typed by their interfaces; that is, the set of services they offer are specified by an interface. For ports to work properly, the interface typing the port must also type the part so that they are compatible. Strictly speaking, the every service offered by the port must also be offered by the realizing part, but the part may offer additional services as well. In practice, it is most common for them to comply to the same interface.

Rhapsody makes it very easy to specify the contract (interface) for the port. Simply double-click on the port to get the features. Once there, you can simply select an interface from the list or click on the Contract tab of the port features dialog. Figure 6 shows that you can select one or more interfaces for both required and provided interfaces. UML 1.x defined the original notion of a provided interface. This uses the standard lollipop notation, as shown in Figure 6. UML 2 adds the idea of a required interface – shown with a socket (see the port on the Display class). A provided interface specifies what services are offered or provided by the port, while a required interface specifies the "other end" of the interface – what it needs. When both ports on different instance or parts are connected, they must be compatible. Most often, this means that they are defined by the *same* interface, but it is *provided* by one port and *required* by the other. That is the case shown in the figure. In general, to be compatible, all the services specified in the required interface *must* appear in the provided interface, however not all services offered must also be required. This means that the provided interface may subclass and extend the required interface, if desired. The advantage of using the ports here is that (1) the Display doesn't rely on any knowledge of the internal parts of the Ventilator – it just invokes the service on the sensorPort of the Ventilator, and (2) for the internal structure of the Ventilator class, we have explicitly stated how the delegation of the request to the Ventilator gets dispatched to the internal part and which internal part handles the request. If desired, we could elide the detail of the internal structure of the Ventilator class on this one diagram and create a second structure diagram for depicting the internal details. The model repository will bind all the information together properly.

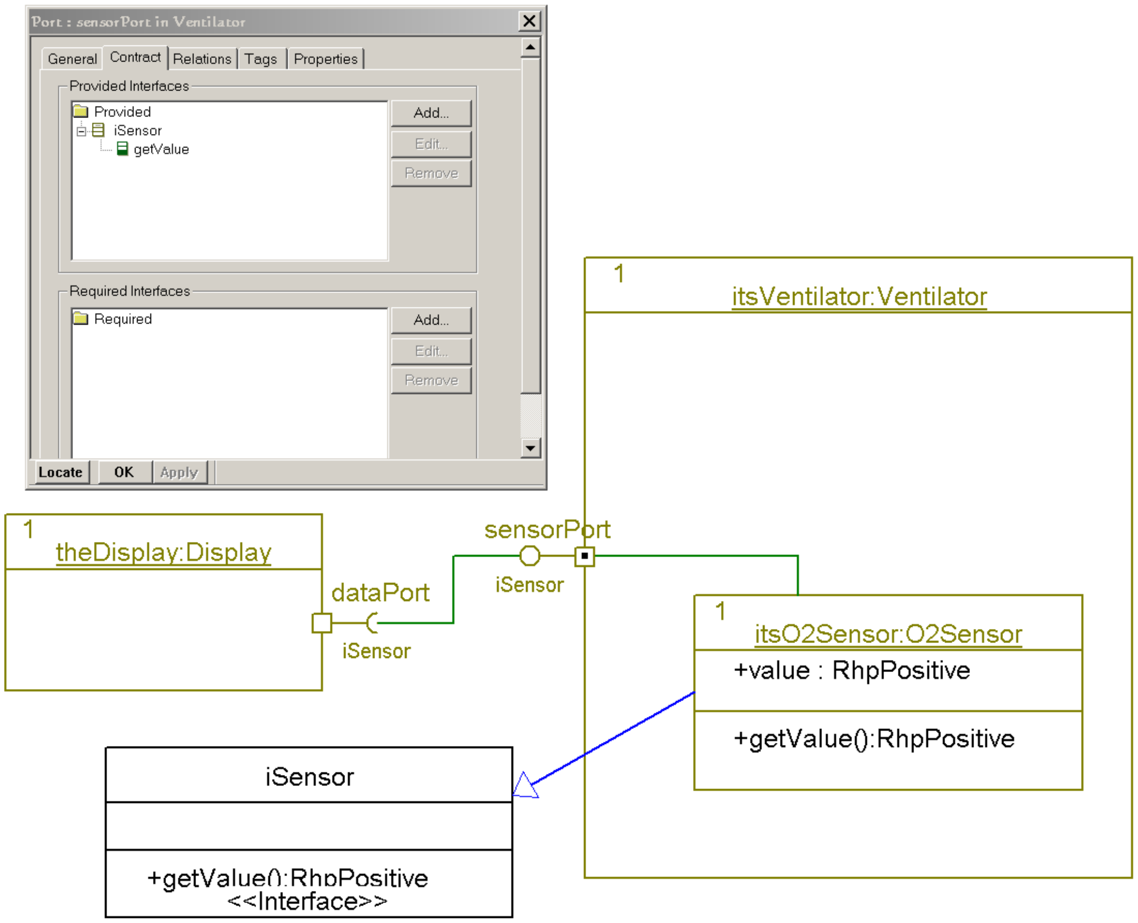


Figure 6: Ports and Contracts

Let's look now at a slightly more complex example. Consider the "port-free" model of the pacing subsystem of a pacemaker, shown in the next figure. Figure 7 shows how we would model such a subsystem in UML 1.x and Rhapsody 4.2.

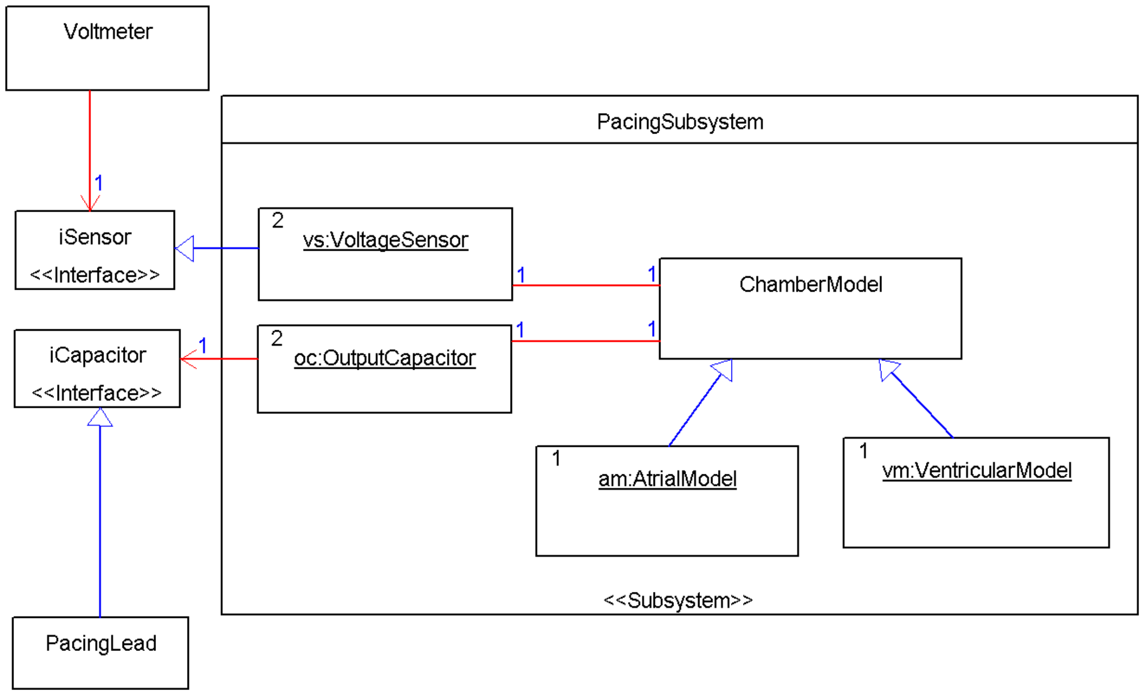


Figure 7: Pacing Subsystem without Ports

There are a couple of interesting points. First, let us suppose that what we really want to model is that the Output Capacitor class *requires* some set of services of the Pacing Lead. Figure 7 doesn't really allow us to do that. What we do instead is show the *provided* interface on the Pacing Lead. Second, notice that there are two VoltageSensor and two OutputCapacitor parts inside the Pacing Subsystem. There are also two Chamber Model instances, one of each subclass. The idea is that the am part connects to the vs part that senses voltage in the atrial chamber of the heart, while the vm part connects to the other vs part that senses voltage in the ventricular chamber. Likewise, the two output capacitors need to be wired to the correct Chamber Model parts. The Pacing Subsystem has this wiring responsibility. How does it do that? It is part of the behavior of the Pacing Subsystem and isn't shown here – it is likely done in the constructor method for the subsystem. Once so wired, the Pacing Subsystem can then run and the pacing chambers sense and pace according to their own behavior models.

How might we do this with ports? It's a little bit more complex than you might think, but only slightly so. Because when you use ports, the wiring of parts is explicit in the structure diagram, you'll need to separate out the instances of the ports. Figure 8 and Figure 9 show the same model with ports and interfaces.

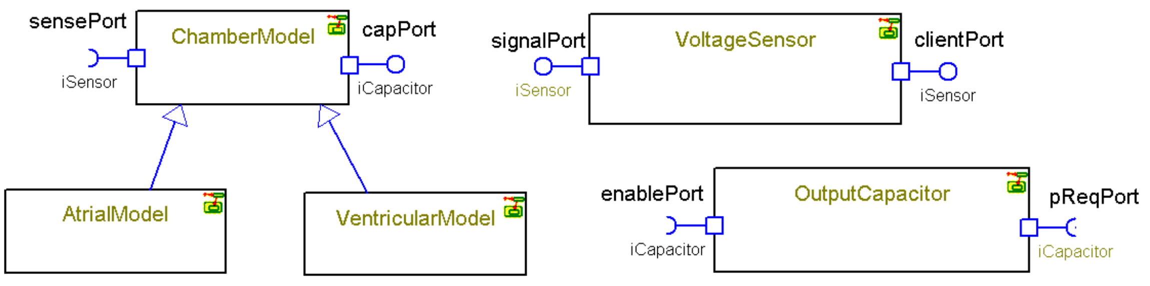


Figure 8: Ports on Classes for Pacer

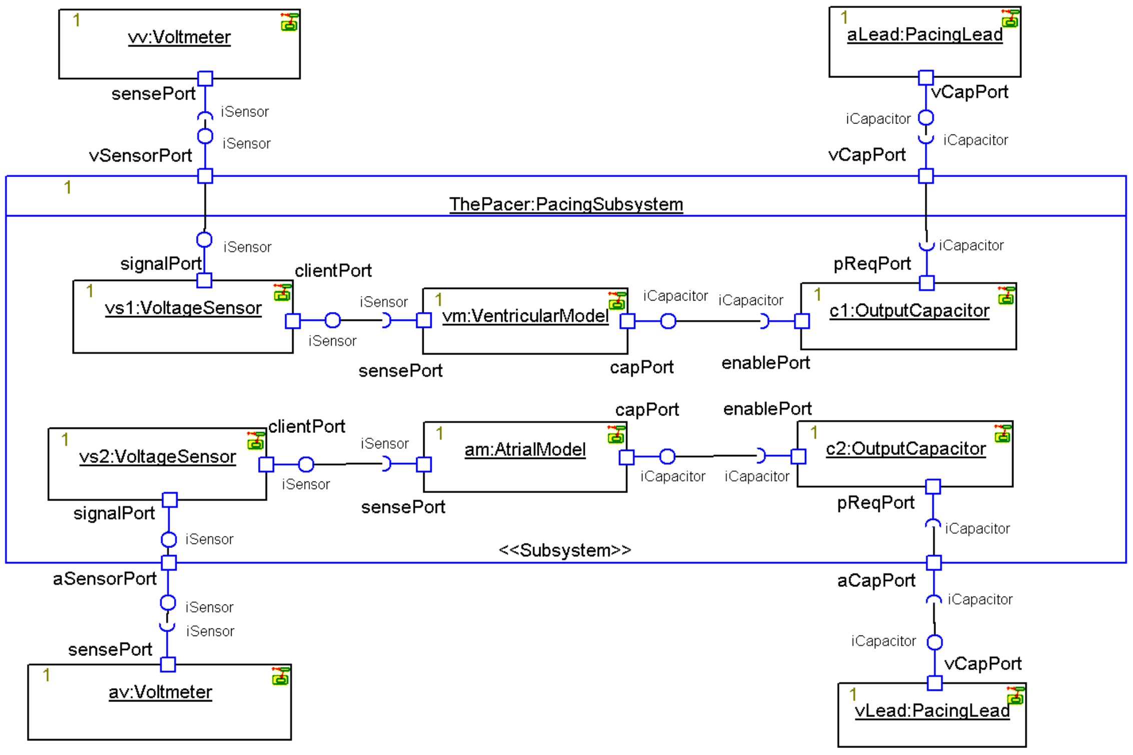


Figure 9: Pacing Subsystem With Ports

What do we notice about the model with ports versus the model without? The most prominent thing is that the model with ports looks a whole lot more complex. There's just more "stuff" on the diagrams. While on one hand, the port version of the model explicitly links together the parts with the interaction points on the subsystem, on the other hand, the model is more complex. Remember when I said that ports are really a design pattern? Design patterns optimize a small set of quality of service (QoS) features at the expense of others, the "pros and cons" of the application of the pattern. The port pattern optimizes the specification of the interaction points at the expense of complexity. If you use ports, then you should make sure that the benefit (specification of the interaction points with the parts) outweighs the increase in complexity. And, by the way, the complexity isn't just in the model, either. The generated code becomes more complex, because each port will result in a class being generated in the source code. The code complexity is mitigated somewhat because if you're using Rhapsody, you're working mostly at the UML model level, but if you do need to work at the code level, this is a potential issue. In some tools, such as RoseRT, you can only connect things together via ports if you want to execute the model. Rhapsody is far more flexible, allowing you to add some interaction points or none, as desired.

More on Ports

Ports are a rich feature and Rhapsody has strong support for various uses of ports. These include the ability to specify behavioral and non-behavioral ports, port multiplicity, a Rhapsody-specific enhancement called *rapid ports*, and, of course, the ability to generate code for ports.

Oh, Behave!

There are two different kinds of ports. *Behavioral ports* are parts of the class that actually implements the behavior. Delegation or relay (non-behavioral) ports relay requests to other ports. Ports may be marked as behavioral by clicking on the Behavioral check box in the features dialog of the port. Figure 10 shows both a delegation and a behavioral port.

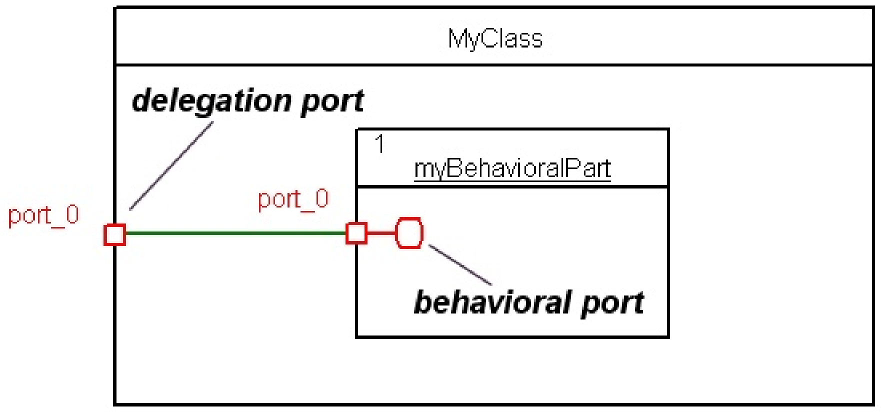


Figure 10: Behavioral Port

Riding the Rapids: Rapid Ports

A port must be specified by its contract, as defined by its provided and required interfaces. Rhapsody provides a special feature that allows you to connect ports together without explicitly specifying the contract. We call this *rapid ports*.
Rapid ports are assumed when ports are drawn and connected and the behavioral port is a port of a class with a statechart. Consider the model in the next figure. Figure 11 shows an instance of a structural class that contains a part called <u>Waxer</u>. The tickTock instance

connects to it via a couple of ports however, none of these have any contracts specified. But because Rhapsody provides rapid ports, they can still collaborate.

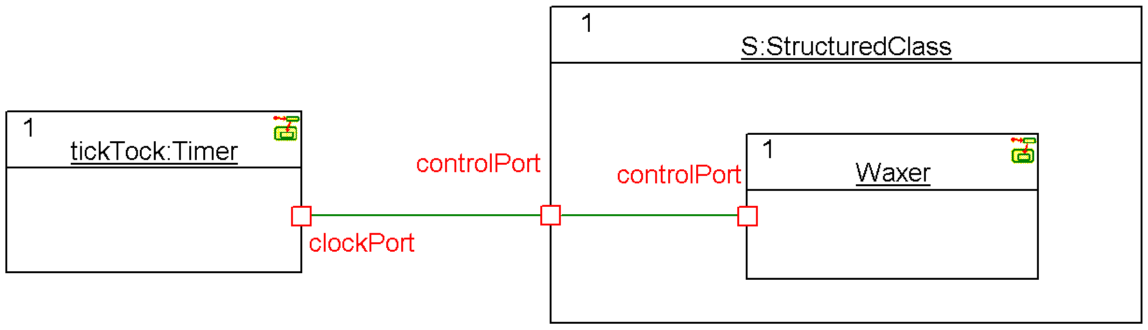


Figure 11: Rapid Ports Example Class Structure

The next two figures show the state machines for the Timer and Waxer classes.

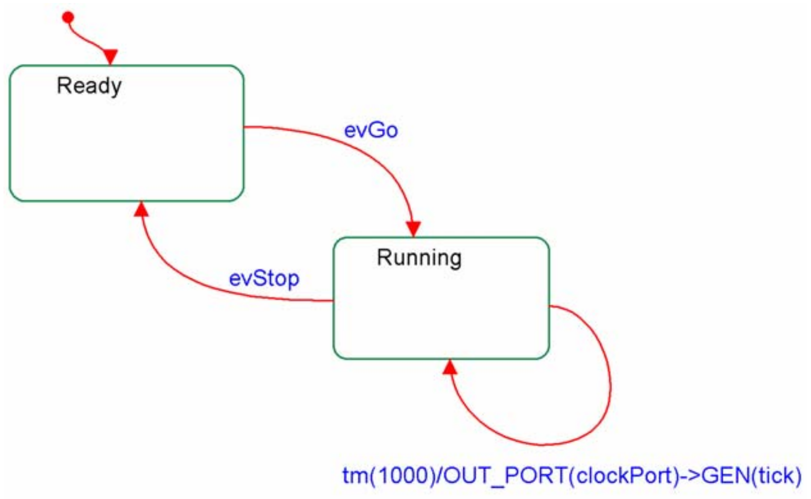


Figure 12:Statechart for Timer

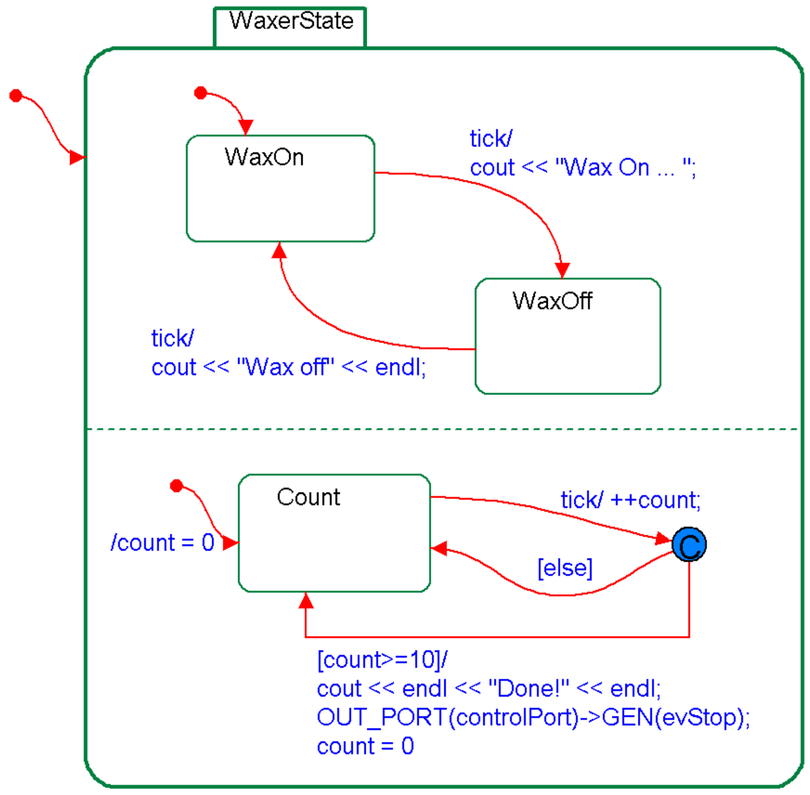


Figure 13: Statechart for Waxer

The OUT_PORT macro in the statecharts allows the named port to be referred to and an event in its contract be invoked. In this case, the OUT_PORT macro is invoking the GEN macro to generate events across the interface.

In the running system, these objects can collaborate across these ports even though we haven't specified the contract. This is called an *implicit contract* and is the key feature of rapid ports.

Port Multiplicity

Just as with parts of a structured class, ports can have multiplicity. The multiplicity is set with the multiplicity field in the port features dialog and is shown on the diagram with the multiplicity inside of square brackets. Just as with association end and part multiplicity, standard multiplicity notation can be used, e.g. '1', '10', '1..10', and '*'.

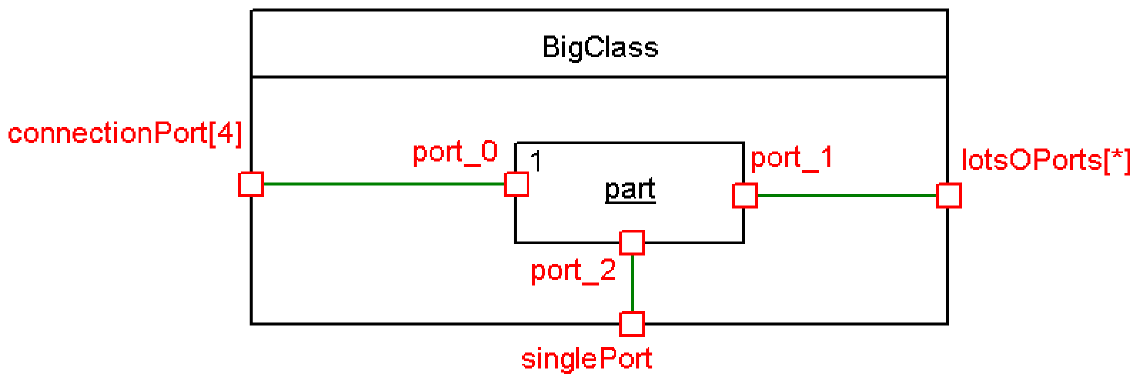


Figure 14: Port Multiplicity

The advantage of port multiplicity is that the receiver of a call or an event can identify which port a call or event came from

Code Generation

Ports are part of the executable model. Rhapsody provides features that allow methods or state machines to access ports, as shown in the table below:

Task	Call
Call an operation f() across a port	OUT_PORT(portName)->f()
Call an operation f() across port #5 of a port with multiplicity 10	OUT_PORT_AT(portName, 5)->f()
Create an event g across a port	OUT_PORT(portName->GEN(g)
Create an event g across port #5 of a port with multiplicity 10	OUT_PORT_AT(portName, 5)->GEN(g)
Listen for an event e from a port as a guard	e[IS_PORT(portName)] / transition actions
Listen for an event e from port #5 of 10, as a guard	e[IS_PORT_AT(portName, 5) / transition actions

The first four of these are used in both operation implementations and as actions on state machines. The last two are used as guards on a transition.

When Rhapsody generates code for a port, the port is realized an "embedded class. The simple case of the BigClass class is shown in Figure 14.

```
//---------------------------------------------------------------------------
// BigClass.h
//---------------------------------------------------------------------------

//## class BigClass
class BigClass : public OMReactive {
public :
        //## class BigClass::part
        class part_C  {
        public :


                //## ignore
                class port_0_C : public OMDefaultReactivePort {
```

```
        ////    Constructors and destructors    ////
        public :
            //## auto_generated
            port_0_C();

            //## auto_generated
            ~port_0_C();

        ////    Operations    ////
        public :
            //## operation connectPart(part_C*)
            void connectPart(part_C* part);


        ////    Attributes    ////
        protected :
            int _p_;         //## attribute _p_

        };

        //## ignore
        class port_1_C : public OMDefaultReactivePort {


        ////    Constructors and destructors    ////
        public :

            //## auto_generated
            port_1_C();

            //## auto_generated
            ~port_1_C();

        ////    Operations    ////
        public :

            //## operation connectPart(part_C*)
            void connectPart(part_C* part);

        ////    Attributes    ////
        protected :

            int _p_;         //## attribute _p_
        };

        //## ignore
        class port_2_C : public OMDefaultReactivePort {


        ////    Constructors and destructors    ////
        public :

            //## auto_generated
            port_2_C();

            //## auto_generated
            ~port_2_C();

        ////    Operations    ////
        public :

            //## operation connectPart(part_C*)
            void connectPart(part_C* part);

        ////    Attributes    ////
        protected :

            int _p_;         //## attribute _p_
        };
//## class BigClass::part
```

```
////    Constructors and destructors    ////
public :
    //## auto_generated
    part_C();

    //## auto_generated
    ~part_C();

////    Additional operations    ////
public :

    //## auto_generated
    port_0_C* getPort_0() const;

    //## auto_generated
    port_0_C* get_port_0() const;

    //## auto_generated
    port_0_C* newPort_0();

    //## auto_generated
    void deletePort_0();

    //## auto_generated
    port_1_C* getPort_1() const;

    //## auto_generated
    port_1_C* get_port_1() const;

    //## auto_generated
    port_1_C* newPort_1();

    //## auto_generated
    void deletePort_1();

    //## auto_generated
    port_2_C* getPort_2() const;

    //## auto_generated
    port_2_C* get_port_2() const;

    //## auto_generated
    port_2_C* newPort_2();

    //## auto_generated
    void deletePort_2();

////    Framework operations    ////
protected :
    //## auto_generated
    void initRelations();

    //## auto_generated
    void cleanUpRelations();

////    Relations and components    ////
protected :
    port_0_C* port_0;         //## classInstance port_0
    port_1_C* port_1;         //## classInstance port_1
    port_2_C* port_2;         //## classInstance port_2

};

//## ignore
class connectionPort_C : public OMDefaultReactivePort {


////    Constructors and destructors    ////
public :
```

```
            //## auto_generated
            connectionPort_C();

            //## auto_generated
            ~connectionPort_C();

        ////    Attributes    ////
        protected :
            int _p_;          //## attribute _p_

        };

        //## ignore
        class lotsOPorts_C : public OMDefaultReactivePort {


        ////    Constructors and destructors    ////
        public :
            //## auto_generated
            lotsOPorts_C();

            //## auto_generated
            ~lotsOPorts_C();


        ////    Attributes    ////
        protected :
            int _p_;          //## attribute _p_

        };

        //## ignore
        class singlePort_C : public OMDefaultReactivePort {

        ////    Constructors and destructors    ////
        public :
            //## auto_generated
            singlePort_C();

            //## auto_generated
            ~singlePort_C();
        ////    Attributes    ////
        protected :

            int _p_;          //## attribute _p_
        };

//## class BigClass


////    Constructors and destructors    ////
public :

    //## auto_generated
    BigClass(OMThread* p_thread = OMDefaultThread);

    //## auto_generated
    ~BigClass();


////    Additional operations    ////
public :

    //## auto_generated
    connectionPort_C* getConnectionPortAt(int i) const;

    //## auto_generated
    connectionPort_C* get_connectionPort(int i) const;

    //## auto_generated
    lotsOPorts_C* getLotsOPortsAt(int i) const;

    //## auto_generated
    lotsOPorts_C* get_lotsOPorts(int i) const;

    //## auto_generated
    lotsOPorts_C* newLotsOPorts();

    //## auto_generated
    void deleteLotsOPorts(lotsOPorts_C* p_lotsOPorts_C);

    //## auto_generated
    singlePort_C* getSinglePort() const;

    //## auto_generated
    singlePort_C* get_singlePort() const;

    //## auto_generated
    part_C* getPart() const;

////    Framework operations    ////
public :

    //## auto_generated
    int getConnectionPort() const;

    //## auto_generated
    OMIterator<lotsOPorts_C*> getLotsOPorts() const;

    //## auto_generated
    virtual OMBoolean startBehavior();

protected :
    //## auto_generated
    void initRelations();

    //## auto_generated
    void cleanUpRelations();

////    Relations and components    ////
protected :

    connectionPort_C connectionPort[4];              //## classInstance connectionPort
    OMList<lotsOPorts_C*> lotsOPorts;        //## classInstance lotsOPorts
    singlePort_C singlePort;         //## classInstance singlePort
    part_C part;              //## classInstance part

};
```

Figure 15: BigClass.h

This is just the header file, but you can see that the part and the ports are embedded classes within the structured class. *partC* is the class of the part named *part* in the figure. Since it was an implicitly typed object, Rhapsody created a class for it when it generated code. Nested inside of that class are the classes for the ports. They all have multiplicity '1' and so are created with simple pointers in the part_c class:

```
protected :
            port_0_C* port_0;        //## classInstance port_0
            port_1_C* port_1;        //## classInstance port_1
            port_2_C* port_2;        //## classInstance port_2

        };
```

The port for the BigClass are created with embedded instances:

```
protected :
    connectionPort_C connectionPort[4];             //## classInstance connectionPort
    OMList<lotsOPorts_C*> lotsOPorts;        //## classInstance lotsOPorts
    singlePort_C singlePort;          //## classInstance singlePort
    part_C part;              //## classInstance part
```

You can also see that ports are subclassed from the OXF framework class OMDefaultReactivePort alleviating you from having to create the delegation behavior by hand. Why "reactive?" Because if you don't specify a contract, Rhapsody provides some default behavior – namely the ability to pass events across the port boundary.

What about ports that are typed by contracts?

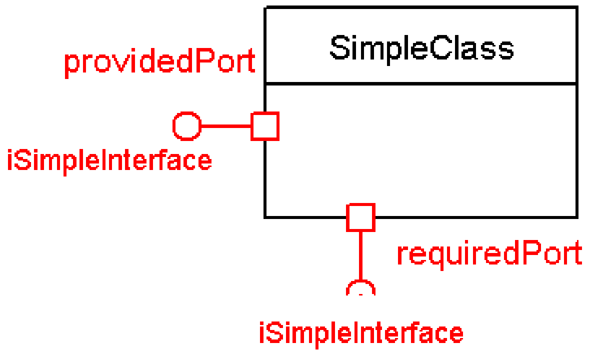


Figure 16: Simple Class

Figure 16 has two ports, one with a required interface and one with a provided interface.

```
//----------------------------------------------------------------------------
// SimpleClass.h
//----------------------------------------------------------------------------
class iSimpleInterface;
class SimpleClass;


//## class SimpleClass
class SimpleClass  {
public :

        //## ignore
        class providedPort_C  {
        public :
            class InBound_C;
            class OutBound_C;

        public :
                //## ignore
                class InBound_C : public iSimpleInterface {

                ////    Constructors and destructors    ////
                public :

                    //## auto_generated
                    InBound_C();

                    //## auto_generated
                    ~InBound_C();

                ////    Operations    ////
                public :

                    //## operation simpleOp()
                    void simpleOp();

                ////    Additional operations    ////
                public :
                    //## auto_generated
                    iSimpleInterface* getItsISimpleInterface() const;

                    //## auto_generated
                    void setItsISimpleInterface(iSimpleInterface* p_iSimpleInterface);

                ////    Framework operations    ////
                protected :

                    //## auto_generated
                    void cleanUpRelations();

                ////    Relations and components    ////
                protected :
                    iSimpleInterface* itsISimpleInterface;  //## link itsISimpleInterface

                };

                //## ignore
                class OutBound_C  {

                ////    Constructors and destructors    ////
                public :

                    //## auto_generated
                    OutBound_C();

                    //## auto_generated
                    ~OutBound_C();

                };

        //## ignore


        ////    Constructors and destructors    ////
        public :

            //## auto_generated
            providedPort_C();

            //## auto_generated
            ~providedPort_C();


        ////    Operations    ////
        public :

            //## operation connectSimpleClass(SimpleClass*)
            void connectSimpleClass(SimpleClass* part);

            //## operation getItsISimpleInterface()
            iSimpleInterface* getItsISimpleInterface();

            //## operation setItsISimpleInterface(iSimpleInterface*)
            void setItsISimpleInterface(iSimpleInterface* p_iSimpleInterface);


        ////    Additional operations    ////
        public :

            //## auto_generated
            InBound_C* getInBound() const;
```

```
            //## auto_generated
            OutBound_C* getOutBound() const;

        ////    Attributes    ////
        protected :
            int _p_;          //## attribute _p_

        ////    Relations and components    ////
        protected :
            InBound_C InBound;               //## classInstance InBound
            OutBound_C OutBound;             //## classInstance OutBound
        };

        //## ignore
        class requiredPort_C  {
        public :
            class InBound_C;
            class OutBound_C;

        public :
                //## ignore
                class InBound_C  {

                ////    Constructors and destructors    ////
                public :
                    //## auto_generated
                    InBound_C();

                    //## auto_generated
                    ~InBound_C();
                };

                //## ignore
                class OutBound_C : public iSimpleInterface {

                ////    Constructors and destructors    ////
                public :

                    //## auto_generated
                    OutBound_C();

                    //## auto_generated
                    ~OutBound_C();

                ////    Operations    ////
                public :

                    //## operation simpleOp()
                    void simpleOp();

                ////    Additional operations    ////
                public :

                    //## auto_generated
                    iSimpleInterface* getItsISimpleInterface() const;

                    //## auto_generated
                    void setItsISimpleInterface(iSimpleInterface* p_iSimpleInterface);

                ////    Framework operations    ////
                protected :

                    //## auto_generated
                    void cleanUpRelations();

                ////    Relations and components    ////
                protected :

                    iSimpleInterface* itsISimpleInterface;  //## link itsISimpleInterface
                };
        //## ignore
        ////    Constructors and destructors    ////
        public :
            //## auto_generated
            requiredPort_C();

            //## auto_generated
            ~requiredPort_C();

        ////    Operations    ////
        public :
            //## operation connectSimpleClass(SimpleClass*)
            void connectSimpleClass(SimpleClass* part);

            //## operation getItsISimpleInterface()
            iSimpleInterface* getItsISimpleInterface();

            //## operation setItsISimpleInterface(iSimpleInterface*)
            void setItsISimpleInterface(iSimpleInterface* p_iSimpleInterface);

        ////    Additional operations    ////
        public :
            //## auto_generated
            InBound_C* getInBound() const;

            //## auto_generated
            OutBound_C* getOutBound() const;


        ////    Attributes    ////
        protected :
            int _p_;          //## attribute _p_

        ////    Relations and components    ////
        protected :
            InBound_C InBound;               //## classInstance InBound
            OutBound_C OutBound;             //## classInstance OutBound
        };

//## class SimpleClass

////    Constructors and destructors    ////
public :
    //## auto_generated
    SimpleClass();

    //## auto_generated
    ~SimpleClass();

////    Additional operations    ////
public :

    //## auto_generated
    providedPort_C* getProvidedPort() const;

    //## auto_generated
    providedPort_C* get_providedPort() const;

    //## auto_generated
    providedPort_C* newProvidedPort();

    //## auto_generated
    void deleteProvidedPort();

    //## auto_generated
    requiredPort_C* getRequiredPort() const;

    //## auto_generated
    requiredPort_C* get_requiredPort() const;
    //## auto_generated
    requiredPort_C* newRequiredPort();

    //## auto_generated
    void deleteRequiredPort();


////    Framework operations    ////
protected :

    //## auto_generated
    void initRelations();

    //## auto_generated
    void cleanUpRelations();

////    Relations and components    ////
protected :
    providedPort_C* providedPort;            //## classInstance providedPort
    requiredPort_C* requiredPort;            //## classInstance requiredPort

};
```

Figure 17: SimpleClass.h file

In this case, the providedPort, which is typed by the iSimpleInterface, is simple (detailed elided):

```
class providedPort_C  {
        public :
            class InBound_C;
            class OutBound_C;

        public :
                //## ignore
                class InBound_C : public iSimpleInterface {
}
        // details elided
        class OutBound_C  {

                        ////    Constructors and destructors    ////
                        public :

                            //## auto_generated
                            OutBound_C();

                            //## auto_generated
                            ~OutBound_C();

                        };
```

We can see the that inbound (provided) part is simply typed by the provided interface class we used and the output (required part is stubbed.

The requiredPort specified a required interface but didn’t specify a provided interface. In this case, the structure is reversed:

```
 class requiredPort_C  {
        public :
            class InBound_C;
            class OutBound_C;

        public :
                //## ignore
                class InBound_C  {

                ////    Constructors and destructors    ////
                public :
                    //## auto_generated
                    InBound_C();

                    //## auto_generated
                    ~InBound_C();
                };

                //## ignore
                class OutBound_C : public iSimpleInterface {
}
```

Of course, the actual operations of the iSimpleInterface need to be overridden so that instead of doing any real behavior, they instead delegate behavior:

```
void SimpleClass::providedPort_C::InBound_C::simpleOp() {
    //#[ operation simpleOp()

    if (itsISimpleInterface != NULL)
        itsISimpleInterface->simpleOp();

    //#]
}
```

SimpleClass must realize the simpleOp behavior and do the actual work when the operation is invoked.

Summary

Structured classes and ports are a rich topic. We could go on for a long time discussing details of code generation, modeling of ports, pros and cons, etc. Hopefully, though, by now you have a pretty good idea about what ports can do for you and, maybe, what they can't. They certainly have their use – they provided named interactions points for structured classes and allow the class (or structure) diagram to depict which part will provide the behavior required of a port. The cost is structural complexity. Whether you elect to use port should depend on whether they lower to total amount of complexity in your model.

Acknowledgements

Thanks to Eldad Palachi and Andy Lapping, two members of our elite I-Logix team for assistance with the models and details for this article.