

Object Based Development Using the UML and C

The UML is rapidly becoming a very hot topic and many companies are realizing the advantages and want to adopt this Object Oriented methodology. However a large number of companies currently use Structured Analysis methods using the C language, and without a lot of training, migrating to adopting UML and the C++ language could be a big risk. Moving to a language such as C++ also imposes higher requirements for memory, which again is a potential problem for cost sensitive systems. An interesting alternative is to move to an Object Based methodology, where by using a sub-set of the UML and continuing to use the C language, the risk is greatly diminished.

SO WHAT EXACTLY IS MEANT BY OBJECT BASED?

The basic concept of the object is relatively easy to understand and adopt, generally C programmers have no real problem with this. The idea of objects communicating via relations is a little less natural and often takes the C programmer a little longer to assimilate and master. On the other hand, the concept of inheritance or generalization as it is known in the UML, is a lot more complicated, and to be able to use it effectively, a lot of training is necessary. The use of virtual inheritance and multiple inheritance can also mean higher demands for memory. So by object based, we really mean using Object Oriented design without inheritance. We can still get the advantages of Object Oriented design such as abstraction and encapsulation yet still keep the size of code to within reasonable limits.

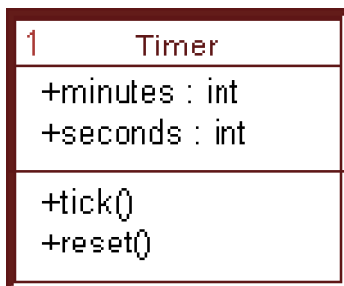


Figure 1.

SO HOW CAN WE USE UML AND THE C LANGUAGE TOGETHER?

Let's take the example of a Timer object, this object could have as responsibility to keep the track of time. It could have attributes such as minutes and seconds, and perhaps methods or operations such as reset and tick. The reset operation could initialize the attributes to zero, and the tick operation could increment the time by one second. In the UML we can create a simple Object Model Diagram such as the following that shows a single object called Timer that has attributes

minutes and seconds of type integer, as well as public operations tick() and reset().

WHAT WILL THE C CODE LOOK LIKE FOR THIS OBJECT?

In C++, then we would see a class. In C, we can use a struct, so this is what the code could look like.

```

struct Timer_t {
    int seconds;
    int minutes;
};

void Timer_tick(struct Timer_t * const me);
void Timer_reset(struct Timer_t * const me);
    
```

The attributes are contained inside the struct, and the operations that can be invoked on these attributes have been prepended with the name of the object. This is what most people would have expected, but what is this argument me that has been added? Well, we could have many Timer objects and we need to be able to distinguish which object the tick and reset operations are to act upon. The argument me is a pointer to a Timer object and can be used by the operations to manipulate the attributes. For those of you who are familiar with C++, this is the "this" pointer, which in C++ is used in exactly the same way as our me pointer, except that the C++ compiler inserts it automatically.

HOW CAN WE CREATE AN INSTANCE OF THIS OBJECT?

Creating an instance statically is very straightforward and can be done as shown:

```

struct Timer_t Timer;
    
```

WHAT ABOUT INITIALIZING THIS OBJECT?

An object can have a constructor and a destructor, how do we handle this in C? well we could add a Timer_Init operation to initialize the object upon creation and a TimerCleanup operation to clean up the

object upon destruction.

```
void Timer_Init (struct Timer_t * const me);
void Timer_Cleanup (struct Timer_t * const me);;
```

WHAT ABOUT IF WE WANT TO DYNAMICALLY CREATE AND DESTROY INSTANCES?

In C++, we can simply call new and delete, in C we could create a Timer_Create operation to create an instance and a Timer_Destroy operation to delete the instance. Note for the Timer_Create operation, that once the memory has been allocated, the Timer_Init operation is invoked.

```
struct Timer_t * Timer_Create() {
    struct Timer_t * me = (struct Timer_t *)
        malloc (sizeof (struct Timer_t));

    Timer_Init(me);
    return me;
}

void Timer_Destroy (struct Timer_t + const me) {
    Timer_Cleanup(me);
    free(me);
}
```

HOW CAN WE WRITE OUR RESET OPERATION?

To do so we have to use the me pointer as follows:

```
void Timer_reset (struct Timer_t * const me) {
    me -> seconds=0;
    me -> minutes=0;
}
```

IN UML, OPERATIONS CAN BE PUBLIC OR PRIVATE, HOW CAN WE ACHIEVE THAT IN C?

The operations that we have shown so far are all public. A private operation is one that can only be invoked from within the object, so in C it makes sense to not declare this operation in the specification (.h) file and to declare it as static in the implementation (.c) file. We also don't need to prepend it with the name of the object, so if we wanted the reset operation to be private, we could use the following declaration:

UML & SDL

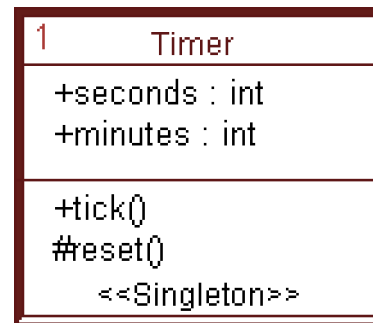


Figure 2.

```
static void reset (struct Timer_t * const me) {
    me -> seconds=0;
    me -> minutes=0;
}
```

WHAT HAPPENS IF I HAVE ONLY ONE INSTANCE OF AN OBJECT, DO I STILL NEED THIS ME POINTER?

We could give the object a stereotype "Singleton" implying that there is only one instance of the object, by doing so, we could then code the reset operation of our Timer object as follows:

```
static void reset() {
    Timer.minutes=0;
    Timer.seconds=0;
}
```

The Timer object is now a global object and the me pointer is no longer necessary, so we can access the attributes directly in the Timer structure as shown.

WHAT ABOUT HANDLING RELATIONS BETWEEN OBJECTS?

A lot can be achieved by using one object, but normally systems are composed of a number of objects that interact together. For two objects to communicate they need to be connected via some sort of relation. For example if we wanted our Timer object to call the show operation of a Display object, then we would add a relation between the Timer and the Display objects. It would make sense to make this a directed association since the communication will be just from the Timer to the Display and not from the Display to the Timer. (See Figure 3)

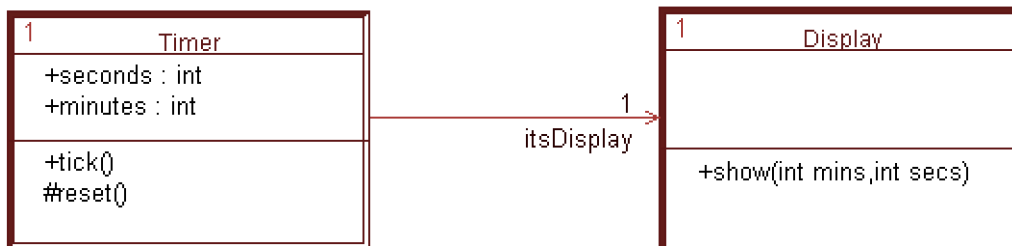


Figure 3.

UML & SDL

The relation will need a role name such as itsDisplay, and a multiplicity to indicate how many Display's the Timer communicates with. In this case it is just one.

HOW WILL A RELATION BE CODED?

When the multiplicity of the relation is one, this can be implemented as a pointer as shown below:

```
struct Timer_t {  
    int seconds;  
    int minutes;  
    struct Display_t * itsDisplay;  
};
```

If the multiplicity is a constant greater than one, then we could use an array of pointers. If the multiplicity is many then perhaps a linked list could be used.

HOW CAN ONE OBJECT CALL ANOTHER OBJECT'S OPERATION?

We have seen that the relation has been coded as a simple pointer to the other object, so in our example, the Timer could call the show operation of the Display object as follows:

```
Display_show (me -> itsDisplay, me -> minutes,  
me -> seconds);
```

HOW COULD WE ADD BEHAVIOR TO AN OBJECT?

In the UML, behavior of an object can be described by using a statechart. For example, we could add a statechart to our Timer object to call the private tick operation every 1000 milliseconds.

The statechart would look like the one on the right: We could also add an "action on entry" to call the show operation on the Display object. That way every second we will increment the time and display it.

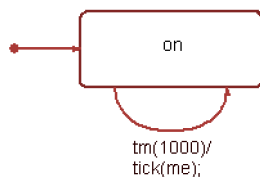


Figure 4.

HOW CAN WE GENERATE CODE WHEN AN OBJECT HAS A STATE-CHART?

This is getting more complicated and it is at this point that we really need to think about building some kind of real-time framework. We need to be able to have some mechanism for handling statecharts and some mechanism for handling timeouts.

```
struct Timer_t {  
    RiCReactive ric_reactive;  
    int seconds;  
    int minutes;  
    struct Display_t * itsDisplay;  
    enum Timer_Enum {Timer_RiCNonState = 0,  
Timer_on = 1} rootState;  
};
```

This real-time framework could contain some reactive object that could be added to our Timer object to basically wait for events or timeouts to occur. Whenever an event or a timeout occurs, it could then call an operation to dispatch the event/timeout. For our example, the code could look like the following:

```
static void dispatchEvent (void * const me, short id)  
{  
    switch (me -> rootState) {  
        case Timer_on:  
            {  
                if (id == Timeout_id) {  
                    tick (me);  
                    me -> rootState = Timer_on;  
                    RiCTask_schedTm ( &me -> ric_reactive,  
1000);  
                }  
                break;  
            }  
        default:  
            break;  
    }  
}
```

WHAT ABOUT EVENTS, HOW CAN THEY BE HANDLED?

An event is generally asynchronous, but sometimes when we need the execution to be more deterministic, events can be synchronous. To see how events are handled, let's modify the Timer statechart to allow the Timer to be started, stopped and reset asynchronously via events. The resulting statechart will look as Figure 5.

The Timer object will now initialize in the off state and then wait for either the event evStartStop or evReset. On receiving the event evStartStop, the object will change from the off state to the on state. In the on state, the tick operation will be executed every second.

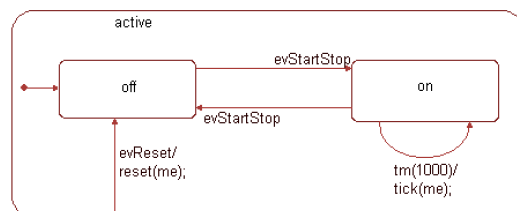


Figure 5.

In this state if the event `evStartStop` is received then the object returns to the off state. In either state, if the event `evReset` occurs, then the reset operation will be executed and the object will result in the off state.

Again a framework needs to be created for handling events. To send an event, a macro could be created so that an event can be sent to an object. To send the event `evStartStop` to our Timer object, the following code could be deployed for another object such as a Button that has a relation to the Timer object, with role name `itsTimer`:

```
CGEN (me -> itsTimer, evStartStop());
```

WHAT ABOUT CONNECTING TO A REAL-TIME OPERATING SYSTEM?

Of course, we often want to be able to make an object (or a group of objects) run on a separate thread. We also want to be able to use Mutexes, Event flags, Semaphores, Message queues etc. We'd also like to be able to make our design independent of any specific RTOS, by using some kind of abstract operating system. This would allow us to use say the Windows OS so that we can rapidly validate our model on the PC. Once validated on the host, we could rapidly re-target for another RTOS to run on the target. An efficient way of handling this is to use a real-time framework. This framework could be constructed so that it is easy to adapt it to another RTOS such as VxWorks, pSOS, OSE, Nucleus, ... or even a custom or proprietary OS.

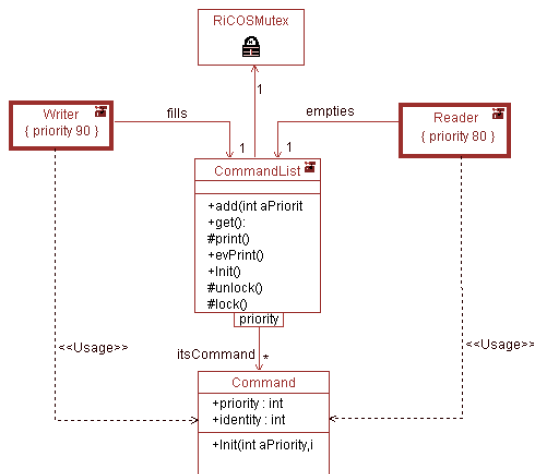


Figure 6.

Figure 6 is an example showing how Threads, Mutexes, ... can be drawn in the UML. Note that in this example, we have used object types rather than objects, this is basically the same as a class. The `Writer` and `Reader` classes are active (thick border) denoting that they run on separate threads. To protect against concurrent access, a mutex has been used.

Figure 7 shows how a group of objects can be run on the same thread.

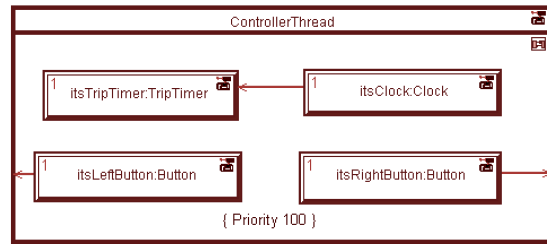


Figure 7.

CAN CODE BE GENERATED AUTOMATICALLY FROM THE MODEL?

Of course, all the code that we have seen so far for the Timer object can be generated automatically. In fact for most models, between 65 and 90% of the code can be generated automatically. The remaining 10 to 35% is code that the programmer writes such as the bodies for the tick and reset operations.

Even from the above diagram, code can be generated automatically for the dependencies, the qualified association, the active classes, the mutex, etc. The programmer just needs to specify the operations and actions on the statecharts.

WHAT ABOUT ACTIVITY DIAGRAMS?

Simple Activity diagrams or flowcharts can be attached to operations for describing the behavior of an operation, so that complex operations can be captured graphically. Before you ask, yes code can be generated from these diagrams as well. Figure 8 shows an example of such a diagram.

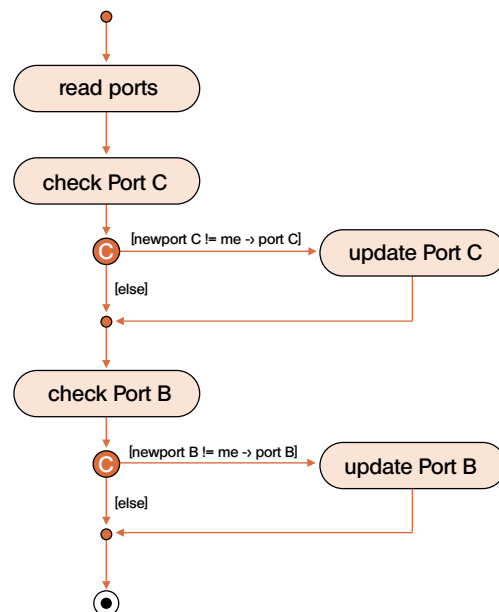


Figure 8.

Code can be generated automatically, but the code for each individual activity must of course still be written by hand.

HOW CAN YOU BE SURE THAT A UML MODEL IS CORRECT?

This is perhaps the most interesting question, you can't really be sure that the model is correct until you have executed the model. Technology is available today to allow graphical back animation. This basically means that code can be generated automatically from the model and instrumented so that when executed, the model is animated. This means that you can validate the model by using the very same diagrams that you used to describe the model. For example, you'll be able to see what instances / objects are currently created, for each object. You'll be able to see the value of the attributes, see what each relation is set to, see what state each object is in, trace the messages sent between objects on a sequence diagram, even step through an activity diagram. This animation can be done at any time during a project and allows the programmer to spend more time being highly productive doing design (the intellectual property) than wasting time doing tedious coding. Problems can be found early on and corrected.

THIS SOUNDS LIKE IT WILL REDUCE THE DEVELOPMENT TIME, IS THAT SO?

Well yes, many companies have already been doing development in this way with the UML and have found that they are reducing the development cycle by at least 30%. Even if you could manage just 10%, on your first project, think of the savings!

BUT WHAT HAPPENS IF THE AUTO GENERATED CODE IS MODIFIED BY HAND?

This is a very common problem, so many times programmers spend time creating a model, then generate the code. Later the code is modified to get it to work and nobody ever has the time or energy to update the model. As time goes on, the model gets more and more out of sync with the code and is less and less useful. Again technology is now available to ensure that any modifications to the code can be "roundtripped" back into the model, ensuring that the code and the model are one and the same thing. This is so important during the maintenance phase and when new features need to be added to a new version.

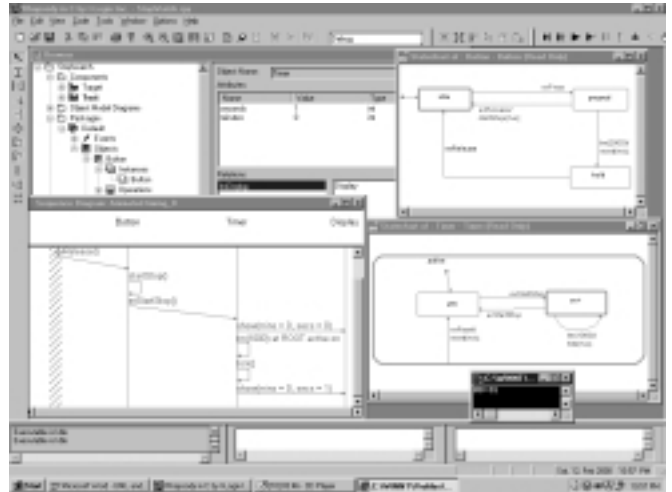


Figure 9.

CAN I GENERATE DOCUMENTATION FROM THE MODEL?

Of course, since the model contains everything, a report can be easily automatically generated according to various standard templates, in word, HTML, framemaker, ...

I HAVE A LOT OF LEGACY CODE, CAN I REUSE THAT?

Everyone has legacy code and of course that can be reused very easily. There are many ways of reusing code, often it is by creating a wrapper object that simply acts as an interface or a façade to the legacy code, it can then be shown on diagrams and the legacy code can be linked in as a library. Here is an example showing how legacy code for a LCD can be added to our Timer example in Figure 10 ■

Mark Richardson is a senior Application Engineer within I-Logix UK. He has more than 15 years experience in designing real-time applications. He now spends most of his time travelling Europe doing consultancy and giving seminars.

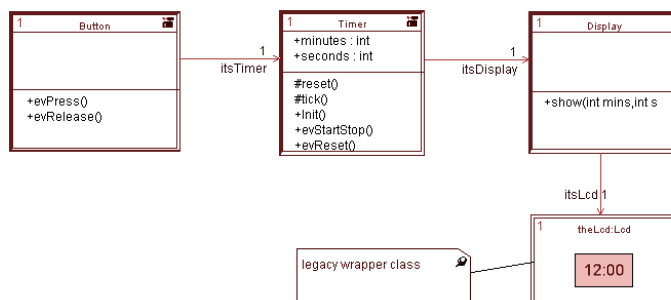


Figure 10.