We verification test bench designers are happy sausage makers, merrily turning out complex and powerful verification environments. To us, object-oriented programming environments not only greatly enhance our productivity, but they make us feel smarter. Who doesn’t like to throw around words such as extend, factory, and, of course, polymorphism. It’s good for the ego and the soul.

However, our test writing coworkers, the ones who will use our environment to drive stimulus into the DUT, look upon object-oriented programming as a horrible goo, that some people need to touch to make the sausages, but that they would rather ignore. When you tell these folks, “Just extend the basic test class, and override the environment in the factory”, they look at you as if you had asked them to plunge their hands into a bowl of freshly ground pork.

In this article, we’ll look at one way of hiding the horrible guts of our work from those with weaker stomachs. We’ll take our elegant object-oriented programming environment and present it to the user as a simple API of SystemVerilog tasks. We’ll do this using a mythical memory that needs to be tested. The memory has a read operation, write operation, an address, and data. Here’s how our test-writing coworkers would like to write a test for this memory:

```
register_test(read_mem)
task read_mem::run_phase(phase);
data_t data;
phase.raise_objection(this);
initialize_memory(.max_addr(10));
for(int ii = 0; ii<=10; ii++) begin: reads
memory(read(.addr(ii), .data(data));
`uvm_info("MEM READ",
$psprintf("Addr: %d Data: %d",ii,data),
UVM_INFO)
end::reads
phase.drop_objection(this);
endtask:run
```

The tasks `initialize_memory()` and `memory_read()` are the API that the test writer uses to access the test bench. What the test writer doesn’t know is that a lot is going on here. The writer has actually created a new `uvm_test` with a factory override in it, extended and agent, and provided an `uvm run()` task that creates sequences and puts them into a sequencer. Let’s look at how this was put together starting at the bottom with a basic memory agent.

**THE MEMORY AGENT**

The memory agent object contains a `driver` (which grabs the interface to the DUT), an `uvm_sequencer`, and a set of methods that the test writer sees as the API. The methods use configurable sequences to send stimulus through the sequencer and into the driver.
**Code 2 : Base Agent Top**

```verilog
class base_memory_agent extends uvm_component;
    `uvm_component_utils(base_memory_agent)
mem_driver drv;
    uvm_sequencer #(mem_req, mem_rsp) seqr;
    mem_sequence seq;
function new(string name, uvm_component parent);
    super.new(name, parent);
endfunction: new;
function void build();
    drv = mem_driver::type_id::create("drv",this);
    seqr = new("seqr", this);
endfunction: build;
function void connect();
    drv.seq_item_port.connect(seqr.seq_item_export);
endfunction: connect
```

We call this class the `base_memory_agent()` because it doesn't actually run the tests, it just provides the test running infrastructure. The top of the agent is straightforward. It provides the driver, sequencer, and a sequence. It also provides the required constructor, `build()` method, and `connect()` method.

Next, we add user API methods to the base agent. Anyone who extends this agent gets access to the tasks.

**Code 3 Bottom of Base Agent**

```verilog
... 
task initialize_memory(int max_addr);
    seq = mem_sequence::type_id::create("seq",this);
    for(int i = 0; i <= max_addr; i++) begin:init
        seq.set_addr(i);
        seq.set_data(i*2);
        seq.set_op(write);
        seq.start(seqr);
    end:init
endtask:initialize_memory

... 
```

In this example we've given our users three API calls: `initialize()`, `memory_write()`, and `memory_read()`.

The API methods all work the same way. They instantiate a sequence, configure it to implement the requested operation and then send the sequence into the sequencer. You can create arbitrarily complex API functions and hide the various sequences from the test writer. You can see the calls to these agents in Code 1.

We'll be extending the agent later in the process.

**THE CONFIGURABLE SEQUENCE**

This method for creating a procedural API to the test-writers requires us to either create a one-to-one mapping between the sequences in our test bench, or to create configurable sequences. In this simple example, we've created configurable sequences.
### Code 4: Top of configurable sequence

```verilog
class mem_sequence extends uvm_sequence #_(mem_req, mem_rsp);
    `uvm_object_utils(mem_sequence)
function new(string name="");
    super.new(name);
endfunction:new

protected mem_req req;
protected mem_rsp rsp;
protected op_t op = nop;
protected addr_t addr = 0;
protected data_t data = 0;

function void set_op(op_t o);
    op = o;
endfunction:set_op

function op_t get_op();
    return op;
endfunction
...
```

Here we have a simple memory sequence with three configurable variables: `addr`, `data`, and `op`. We configure the variables using the set/get accessor approach for all three variables.

### Code 5: Sequence Body

```verilog
...  function data_t get_data();
      return data;
  endfunction

task body;
    req = new();
    start_item(req);
    req.op = op;
    req.addr = addr;
    req.data = data;
    finish_item(req);
    if(req.op == read) begin:read_data
        get_response(rsp);
        data = rsp.data;
    end:read_data
endtask:body
endclass:mem_sequence
```

The sequence body is very simple in this case. It creates a sequence item, loads it with data, and sends it into the sequencer. Then it reads the response if this is a read operation.

Though this is a simple example, you can make sequences such as this arbitrarily complex. For example, this could be a virtual sequence that orchestrates a wide variety of operations across the test bench. The test writer would be concealed from all this complexity by the API.

### The Memory Environment

We’re now ready to stitch the agent together with an `uvm_test` to create the simple system from Code 1. Here is the environment:

### Code 6: The Memory Environment

```verilog
class mem_env extends uvm_env;
    `uvm_component_utils(mem_env);
base_memory_agent agent;

function new(string name, uvm_component parent);
    super.new(name, parent);
endfunction:new

function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    agent = base_memory_agent::type_id::create("agent", this);
endfunction: build
endclass:mem_env
```

This simple environment only has one job. It declares the `base_memory_agent` as an object called `agent`, then it
instantiates the agent using the factory. It’s critical that we use the factory here, as we will see when we write the top level uvm_test.

THE BASIC TEST

The test object instantiates the basic environment that we defined above.

**Code 7:**

```markdown
class mem_test extends uvm_test;
  `uvm_component_utils(mem_test);
  mem_env env;
  function new(string name, uvm_component parent);
    super.new(name, parent);
  endfunction:new
  function void build_phase(phase);
    super.build_phase(phase);
    env = mem_env::type_id::create("env",this);
  endfunction:build_phase
endclass:mem_test
```

This test becomes the base class for all the tests in the test bench. It’s only job is to instantiate the environment that instantiates the base agent.

TYING IT ALL TOGETHER FOR THE TEST WRITER

Now we’re ready to tie all these pieces together for the test writer. Let’s look back at the basic test from Code 1:

**Code 1 Again: the Basic Test**

```markdown
`register_test(read_mem)
 task read_mem::run_phase(phase);
   data_t data;
   phase.raise_objection(this);
   initialize_memory(.max_addr(10));
   for(int ii = 0; ii<=10; ii++) begin: reads
     memory(read(.addr(ii), .data(data));
     `uvm_info("MEM READ",
``````

The key to this approach is the macro at the top of the file. The `register_test` macro allows the test writer to do the following object oriented operations without knowing it:

Extend the `base_mem_agent` to create a new agent called `read_mem`. The new `read_mem` agent defined an external `run()` task that the user will supply.

Extend the `mem_test` to create a new test called `read_mem_test`. This test overrides the `base_mem_agent` with `read_mem` in the factory, then it calls `super.build()` `mem_test` instantiates the `mem_env`.

The override causes the `mem_env` to get a `read_mem` agent instead of a `base_mem_agent`, and the `read_mem` agent can run the test.

The user supplies a `run()` method for the `read_mem` agent, and that `run()` method calls the API tasks that were defined in the `base_mem_agent` base class.

Our user was able to implement all sorts of hidden OOP functionality because of the `register_test` macro.

THE REGISTER TEST MACRO

The register test macro uses the test name to create a new `uvm_test` and a new agent with an externally defined run task. Here is the macro:

```markdown`
ifndef TEST_MACRO
`define TEST_MACRO
`define register_test(__testname) \
class __testname extends base_memory_agent;\`
  `uvm_component_utils(__testname)\`
  function new(string name = "", uvm_component parent
``````
The macro takes the argument __testname and uses it to create a new agent with an externally defined run_phase() task, and a new mem_test that overrides the base_memory_agent.

Notice that the build_phase() method in the __testname_test overrides the factory first, and then calls super.build_phase() so that mem_test can build the environment.

This macro allows the user to focus on writing the run script.

Notice also that the user sees a consistent name such as read_mem in the test file. The user defines the name of the test and then uses that name in the run task definition. The name read_mem_test gets used when you run the simulation with the UVM_TESTNAME option.

Here is another example of the user's point of view:

write test for the same DUT:

**Code 9: A Write Test using the same DUT and Test Bench**

```vhdl
register_test(write_mem)
task write_mem::run_phase(uvm_phase phase);
    phase.raise_objection(this);
    for (int ii = 0; ii <= 5; ii++) begin
        memory_write(.addr(ii), .data(ii));
        for (int ii = 0; ii <= 5; ii++) begin
            memory_read(.addr(ii), .data(data));
            `uvm_info("MEM READ", $psprintf("Addr: %d Data: %d", ii, data), UVM_INFO)
            end : read_loop
        end : read_loop
        phase.drop_objection(this);
    endtask : run
```

Here the user has written a write test and has not had to modify any aspect of the test bench. All the user needs to do is include this file in a package so it gets compiled and run VSIM with the +UVM_TESTNAME=write_mem_test.

**SUMMARY**

Most test writers don’t want to see the OOP guts that make up their sausages. Instead, they want to work with a simple programming interface that drives stimulus into their designs. In this article we examined one approach for allowing users to use tasks to create sequences and drive them into their test bench. We’ve seen that we can hide OOP features such as extension, factory overrides, and polymorphism, and provide test writers with the simple, tasty, meat and potatoes that they know and love.