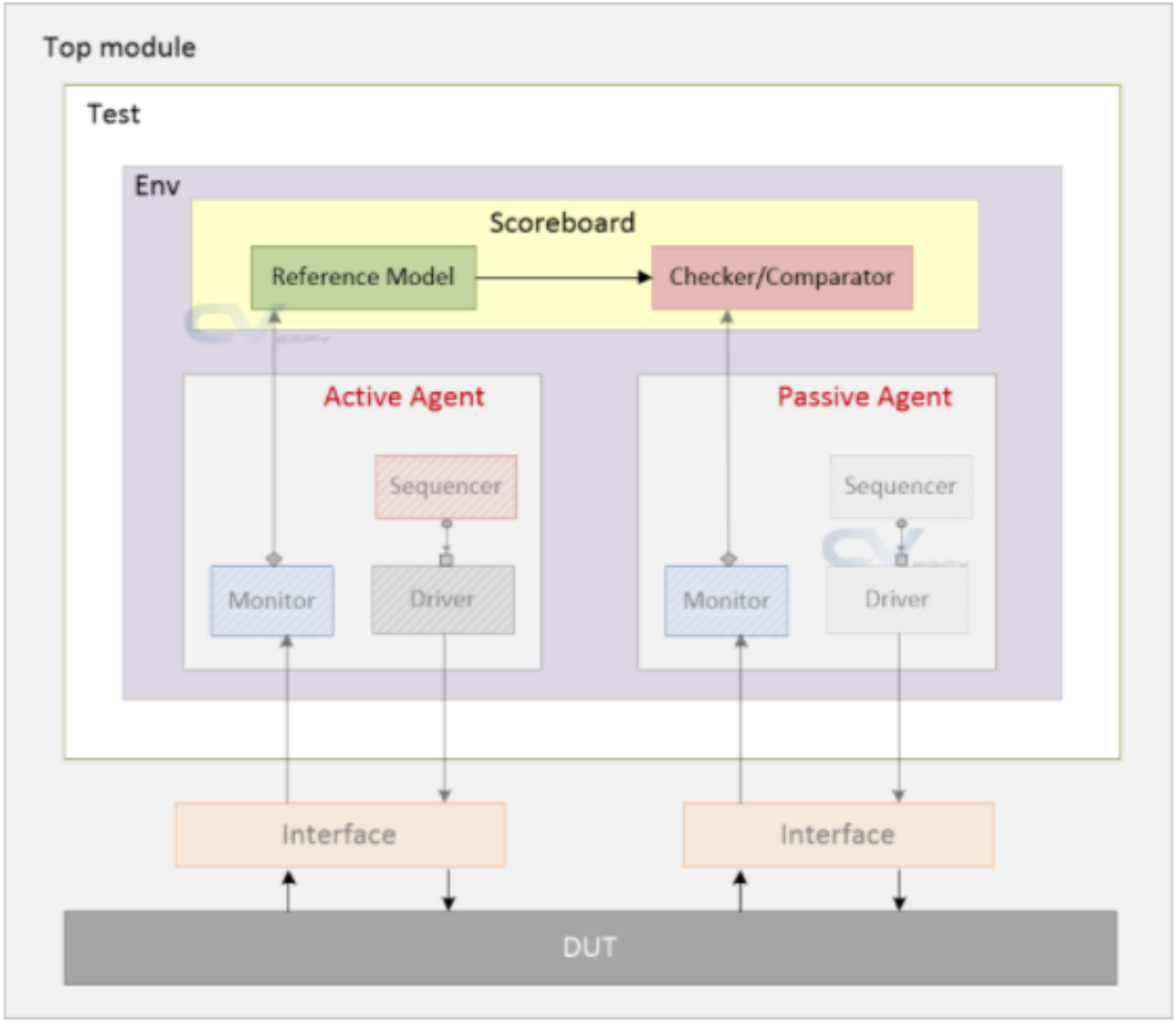
UVM Scoreboard



Creating a UVM Scoreboard

Step 1: Create a custom class inherited from uvm_scoreboard, register with factory, and call function new

```
// my_scoreboard is user-given name for this class that has been derived from "uvm_scoreboard"
class my_scoreboard extends uvm_scoreboard;

    // [Recommended] Makes this scoreboard more re-usable
    `uvm_component_utils (my_scoreboard)

    // This is standard code for all components
    function new (string name = "my_scoreboard", uvm_component parent = null);
      super.new (name, parent);
    endfunction

    // Code for rest of the steps come here
endclass
```

Creating a UVM Scoreboard

2. Add necessary TLM exports to receive transactions from other components. Instantiate them in build_phase

```
// Step2: Declare and create a TLM Analysis Port to receive data objects from other TB components
uvm_analysis_imp #(apb_pkt, my_scoreboard) ap_imp;

// Instantiate the analysis port, because afterall, its a class object
function void build_phase (uvm_phase phase);
    ap_imp = new ("ap_imp", this);
endfunction
```

3. Define the action to be taken when data is received from analysis port

```
// Step3: Define action to be taken when a packet is received via the declared analysis port
virtual function void write (apb_pkt data);
    // What should be done with the data packet received comes here - let's display it
    `uvm_info ("write", $sformatf("Data received = 0x%0h", data), UVM_MEDIUM)
endfunction
```

Creating a UVM Scoreboard

4. Perform checks

```
// Step4: [Optional] Perform any remaining comparisons or checks before end of simulation
virtual function void check_phase (uvm_phase phase);
    ...
endfunction
```

5. Connect analysis port of scoreboard with other components in the environment

```
class my_env extends uvm_env;
    ...

    // Step5: Connect the analysis port of the scoreboard with the monitor so that
    // the scoreboard gets data whenever monitor broadcasts the data.
    virtual function void connect_phase (uvm_phase phase);
        super.connect_phase (phase);
        m_apb_agent.m_apb_mon.analysis_port.connect (m_scbd.ap_imp);
    endfunction
endclass
```

```
// Step1 : Create a new class that extends from uvm_scoreboard
class my_scoreboard extends uvm_scoreboard;
    `uvm_component_utils (my_scoreboard)

    function new (string name = "my_scoreboard", uvm_component parent);
        super.new (name, parent);
    endfunction

    // Step2a: Declare and create a TLM Analysis Port to receive data objects from other TB components
    uvm_analysis_imp #(apb_pkt, my_scoreboard) ap_imp;

    // Step2b: Instantiate the analysis port, because afterall, its a class object
    function void build_phase (uvm_phase phase);
        ap_imp = new ("ap_imp", this);
    endfunction

    // Step3: Define action to be taken when a packet is received via the declared analysis port
    virtual function void write (apb_pkt data);
        // What should be done with the data packet received comes here - let's display it
        `uvm_info ("write", $sformatf("Data received = 0x%0h", data), UVM_MEDIUM)
    endfunction

    // Step3: Define other functions and tasks that operate on the data and call them
    // Remember, this is the main task that consumes simulation time in UVM
    virtual task run_phase (uvm_phase phase);
        ...
    endtask

    // Step4: [Optional] Perform any remaining comparisons or checks before end of simulation
    virtual function void check_phase (uvm_phase phase);
        ...
    endfunction
endclass
```

Connecting UVM Analysis ports of UVM Scoreboard

```
class my_env extends uvm_env;
    `uvm_component_utils (my_env)

    function new (string name = "my_env", uvm_component parent);
        super.new (name, parent);
    endfunction

    // Declare a handle so that we can connect TB components to this
    my_scoreboard   m_scbd;

    // Instantiate or Build the scoreboard using standard UVM factory create calls
    virtual function void build_phase (uvm_phase phase);
        super.build_phase (phase);
        m_scbd = my_scoreboard::type_id::create ("m_scbd", this);
    endfunction

    // Step5: Connect the analysis port of the scoreboard with the monitor so that
    // the scoreboard gets data whenever monitor broadcasts the data.
    // Note: This agent is assumed to be present in this environment for example purpose
    virtual function void connect_phase (uvm_phase phase);
        super.connect_phase (phase);
        m_apb_agent.m_apb_mon.analysis_port.connect (m_scbd.ap_imp);
    endfunction
endclass
```

UVM Environment

1. Code the environment class by extending uvm_env

```
class mem_model_env extends uvm_env;

  `uvm_component_utils(mem_model_env)

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

endclass : mem_model_env
```

2. Declare the agent

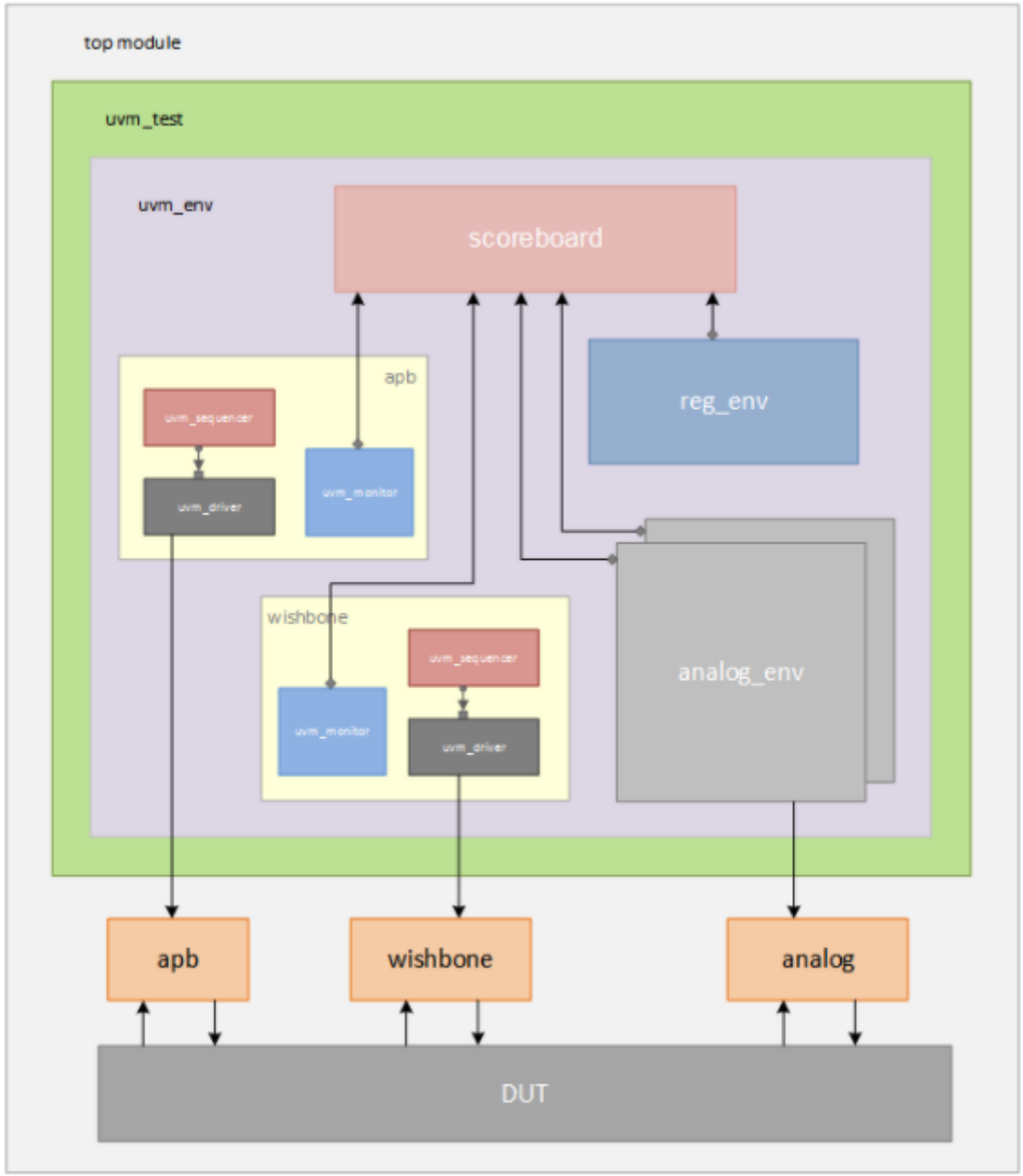
```
mem_agent mem_agnt;
```

3. Create the agent

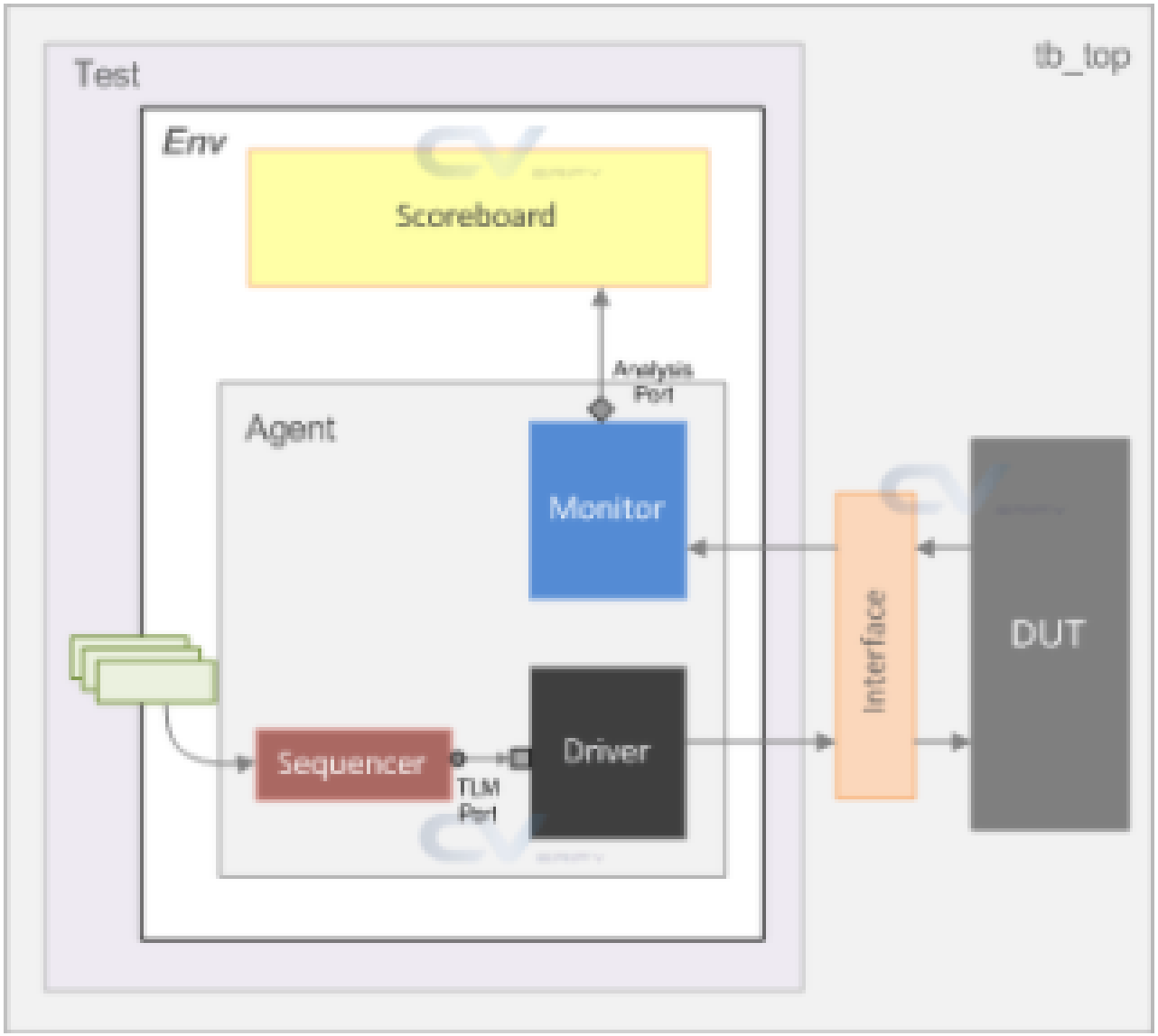
```
mem_agnt = mem_agent::type_id::create("mem_agnt", this);
```

UVM Environment

```
class mem_model_env extends uvm_env;

  mem_agent mem_agnt;

  `uvm_component_utils(mem_model_env)

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

  // build_phase
  function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    mem_agnt = mem_agent::type_id::create("mem_agnt", this);
  endfunction : build_phase

endclass : mem_model_env
```


top module
uvm_test
uvm_env
scoreboard
reg_env
apb
uvm_sequencer
uvm_driver
uvm_monitor
wishbone
uvm_sequencer
uvm_monitor
uvm_driver
analog_env
apb
wishbone
analog
DUT

```
class my_top_env extends uvm_env;
   `uvm_component_utils (my_env)

   agent_apb          m_apb_agt;
   agent_wishbone     m_wb_agt;

   env_register       m_reg_env;
   env_analog         m_analog_env [2];

   scoreboard         m_scbd;

   function new (string name = "my_env", uvm_component parent);
      super.new (name, parent);
   endfunction

   virtual function void build_phase (uvm_phase phase);
      super.build_phase (phase);
      // Instantiate different agents and environments here
      m_apb_agt = agent_apb::type_id::create ("m_apb_agt", this);
      m_wb_agt  = agent_wishbone::type_id::create ("m_wb_agt", this);

      m_reg_env = env_register::type_id::create ("m_reg_env", this);
      foreach (m_analog_env[i])
        m_analog_env[i] = env_analog::type_id::create ($sformatf("m_analog_env%0d",m_analog_env[.]), this);

      m_scbd = scoreboard::type_id::create ("m_scbd", this);
   endfunction

   virtual function void connect_phase (uvm_phase phase);
       // Connect between different environments, agents, analysis ports, and scoreboard here
   endfunction
endclass
```


tb_top
Test
Env
Scoreboard
Analysis
Port
Agent
Monitor
Interface
DUT
Sequencer
TLM
Port
Driver

```
class my_env extends uvm_env ;

   `uvm_component_utils (my_env)

   my_agent             m_agnt0;
   my_scoreboard        m_scbd0;

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

   virtual function void build_phase (uvm_phase phase);
      super.build_phase (phase);
      m_agnt0 = my_agent::type_id::create ("my_agent", this);
      m_scbd0 = my_scoreboard::type_id::create ("my_scoreboard", this);
   endfunction : build_phase

   virtual function void connect_phase (uvm_phase phase);
      // Connect the scoreboard with the agent
      m_agnt0.m_mon0.item_collected_port.connect (m_scbd0.data_export);
   endfunction

endclass
```

UVM Test

1. Code the test by extending uvm_test

```
// Step 1: Declare a new class that derives from "uvm_test"
// my_test is user-given name for this class that has been derived from "uvm_test"
class my_test extends uvm_test;

    // [Recommended] Makes this test more re-usable
    `uvm_component_utils (my_test)

    // This is standard code for all components
    function new (string name = "my_test", uvm_component parent = null);
      super.new (name, parent);
    endfunction

    // Code for rest of the steps come here
endclass
```

UVM Test

2. Declare other environments & verification components and build them

```
my_env   m_top_env;              // Testbench environment that contains other agents, register models,
my_cfg   m_cfg0;                 // Configuration object to tweak the environment for this test

// Instantiate and build components declared above
virtual function void build_phase (uvm_phase phase);
   super.build_phase (phase);

   // [Recommended] Instantiate components using "type_id::create()" method instead of new()
   m_top_env  = my_env::type_id::create ("m_top_env", this);
   m_cfg0     = my_cfg::type_id::create ("m_cfg0", this);

   // [Optional] Configure testbench components if required, get virtual interface handles, etc
   set_cfg_params ();

   // [Recommended] Make the cfg object available to all components in environment/agent/etc
   uvm_config_db #(my_cfg) :: set (this, "m_top_env.my_agent", "m_cfg0", m_cfg0);
endfunction
```

UVM Test

3. Print UVM topology (useful for debug purposes)

```
virtual function void end_of_elaboration_phase (uvm_phase phase);
   uvm_top.print_topology ();
endfunction
```

4. Start a virtual sequence

```
// Start a virtual sequence or a normal sequence for this particular test
virtual task run_phase (uvm_phase phase);

    // Create and instantiate the sequence
    my_seq m_seq = my_seq::type_id::create ("m_seq");

    // Raise objection - else this test will not consume simulation time*
    phase.raise_objection (this);

    // Start the sequence on a given sequencer
    m_seq.start (m_env.seqr);

    // Drop objection - else this test will not finish
    phase.drop_objection (this);
endtask
```

UVM Test

How to run a UVM test?

- A test is started within the testbench_top by a task called run_test
- This is a global task that must be supplied with name of user-defined UVM test to be run

```
// Specify the testname as an argument to the run_test () task
initial begin
   run_test ("base_test");
end
```

UVM Test

How to run ANY UVM test?

```
// Pass the DEFAULT test to be run if nothing is provided through command-line
initial begin
   run_test ("base_test");
   // Or you can leave the argument as blank
   // run_test ();
end

// Command-line arguments for an EDA simulator
$> [simulator] -f list +UVM_TESTNAME=base_test
```

```
// Step 1: Declare a new class that derives from "uvm_test"
class base_test extends uvm_test;

   // Step 2: Register this class with UVM Factory
  `uvm_component_utils (base_test)

  // Step 3: Define the "new" function
  function new (string name, uvm_component parent = null);
     super.new (name, parent);
  endfunction

  // Step 4: Declare other testbench components
  my_env   m_top_env;              // Testbench environment
  my_cfg   m_cfg0;                 // Configuration object


  // Step 5: Instantiate and build components declared above
  virtual function void build_phase (uvm_phase phase);
     super.build_phase (phase);

     // [Recommended] Instantiate components using "type_id::create()" method instead of new()
     m_top_env  = my_env::type_id::create ("m_top_env", this);
     m_cfg0     = my_cfg::type_id::create ("m_cfg0", this);

     // [Optional] Configure testbench components if required
     set_cfg_params ();

     // [Optional] Make the cfg object available to all components in environment/agent/etc
     uvm_config_db #(my_cfg) :: set (this, "m_top_env.my_agent", "m_cfg0", m_cfg0);
  endfunction

  // [Optional] Define testbench configuration parameters, if its applicable
  virtual function void set_cfg_params ();
     // Get DUT interface from top module into the cfg object
     if (! uvm_config_db #(virtual dut_if) :: get (this, "", "dut_if", m_cfg0.vif)) begin
        `uvm_error (get_type_name (), "DUT Interface not found !")
     end
```

```
    // Assign other parameters to the configuration object that has to be used in testbench
    m_cfg0.m_verbosity    = UVM_HIGH;
    m_cfg0.active         = UVM_ACTIVE;
  endfunction

    // [Recommended] By this phase, the environment is all set up so its good to just print the topology for debug
  virtual function void end_of_elaboration_phase (uvm_phase phase);
    uvm_top.print_topology ();
  endfunction

  function void start_of_simulation_phase (uvm_phase phase);
    super.start_of_simulation_phase (phase);

    // [Optional] Assign a default sequence to be executed by the sequencer or look at the run_phase ...
    uvm_config_db#(uvm_object_wrapper)::set(this,"m_top_env.my_agent.m_seqr0.main_phase",
                                    "default_sequence", base_sequence::type_id::get());

  endfunction

  // or [Recommended] start a sequence for this particular test
  virtual task run_phase (uvm_phase phase);
   my_seq m_seq = my_seq::type_id::create ("m_seq");

   super.run_phase(phase);
   phase.raise_objection (this);
   m_seq.start (m_env.seqr);
   phase.drop_objection (this);
  endtask
endclass
```

Derivative Tests

```
// Build a derivative test that launches a different sequence
// base_test <- dv_wr_rd_register_test
class dv_wr_rd_register_test extends base_test;
    `uvm_component_utils (dv_wr_rd_register_test)

    function new(string name = "dv_wr_rd_register_test");
        super.new(name);
    endfunction

    // Start a different sequence for this test
    virtual task run_phase(uvm_phase phase);
        wr_rd_reg_seq   m_wr_rd_reg_seq = wr_rd_reg_seq::type_id::create("m_wr_rd_reg_seq");

        super.run_phase(phase);
        phase.raise_objection(this);
        m_wr_rd_reg_seq.start(m_env.seqr);
        phase.drop_objection(this);
    endtask
endclass
```

Derivative Tests

```
// Build a derivative test that builds a different configuration
// base_test <- dv_wr_rd_register_test <- dv_cfg1_wr_rd_register_test

class dv_cfg1_wr_rd_register_test extends dv_wr_rd_register_test;
    `uvm_component_utils (dv_cfg1_wr_rd_register_test)

    function new(string name = "dv_cfg1_wr_rd_register_test");
        super.new(name);
    endfunction

    // First calls base_test build_phase which sets m_cfg0.active to ACTIVE
    // and then here it reconfigures it to PASSIVE
    virtual function void build_phase(uvm_phase phase);
        super.build_phase(phase);
        m_cfg0.active = UVM_PASSIVE;
    endfunction
endclass
```

UVM Testbench Top

- Testbench top is the module that connects the DUT and the Verification environment components.

- It contains:
  - **DUT instance**
  - **Interface instance**
  - **run_test method**
  - **Virtual interface set_config_db**
  - **Clock and reset generation logic**
  - **Wave dump logic**

```
module tb_top;
   import uvm_pkg::*;

   // Complex testbenches will have multiple clocks and hence multiple clock
   // generator modules that will be instantiated elsewhere
   // For simple designs, it can be put into testbench top
   bit clk;
   always #10 clk <= ~clk;


   // Instantiate the Interface and pass it to Design
   dut_if         dut_if1  (clk);
   dut_wrapper    dut_wr0  (._if (dut_if1));


   // At start of simulation, set the interface handle as a config object in UVM
   // database. This IF handle can be retrieved in the test using the get() method
   // run_test () accepts the test name as argument. In this case, base_test will
   // be run for simulation
   initial begin
      uvm_config_db #(virtual dut_if)::set (null, "uvm_test_top", "dut_if", dut_if1);
      run_test ("base_test");
   end

   // Multiple EDA tools have different system task calls to specify and dump waveform
   // in a given format or path. Some do not need anything to be placed in the testbench
   // top module. Lets just dump a very generic waveform dump file in *.vcd format
   initial begin
        $dumpvars;
        $dumpfile("dump.vcd");
   end

endmodule
```

UVM Class diagram - Review