- UVM Driver
- UVM Sequencer
- UVM Monitor
- UVM Agent

Interface UVCs

- UVM Driver, UVM Sequencer
  - Explain Stimulus modeling and generation
  - Give examples for Stimulus modeling and generation
  - Describe creating the driver and sequencer
  - Give examples for creating the driver and sequencer

Automating UVC Creation

- UVM Monitor, UVM Agent
  - Describe creating the collector and monitor, modeling topology with UVM, creating the Agent
  - Explain creating the UVM verification component, creating UVM sequences, configuring the sequencer's default sequence
  - Apply coordinating end-of-test, implementing protocol-specific coverage and checks, handling reset, packaging interface UVCs

UVM Driver

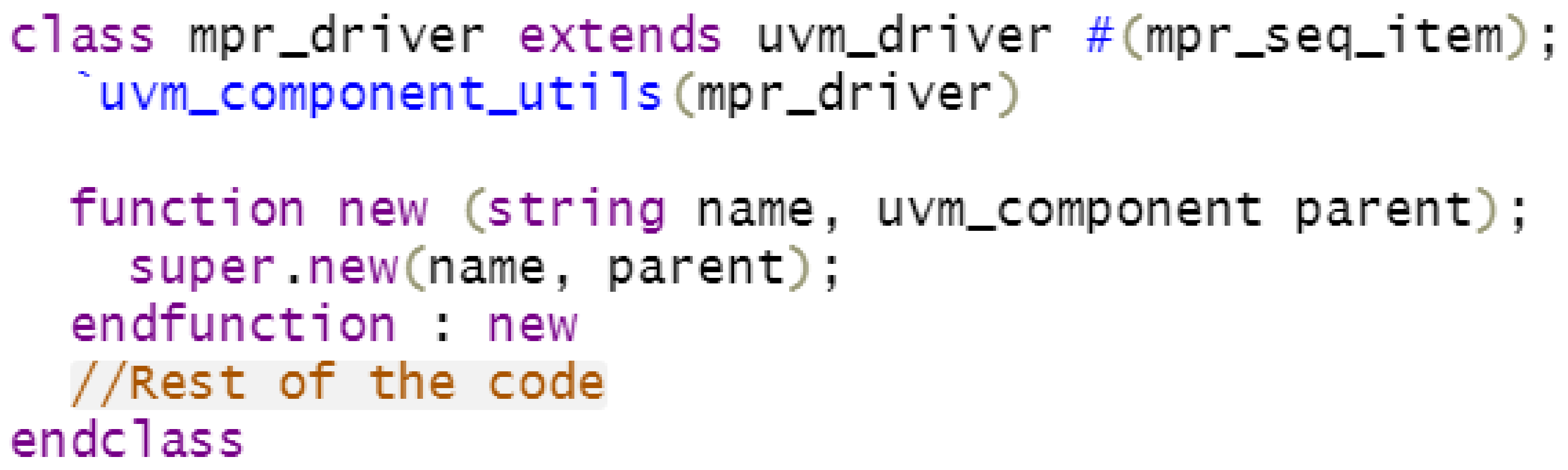
- A driver is written by extending the uvm_driver
- uvm_driver is inherited from uvm_component
  - Methods and TLM port (seq_item_port) are defined for communication between sequencer and driver
- The uvm_driver is a parameterized class
  - parameterized with the type of the request sequence_item and the type of the response sequence_item

UVM Driver Methods

- get_next_item - blocks until a REQ sequence_item is available in the sequencer
- try_next_item - non-blocking variant of the get_next_item() method
- item_done - completes the driver-sequencer handshake and it should be called after a get_next_item() or a successful try_next_item() call
- put - non-blocking and is used to place an RSP sequence_item in the sequencer.

UVM Driver

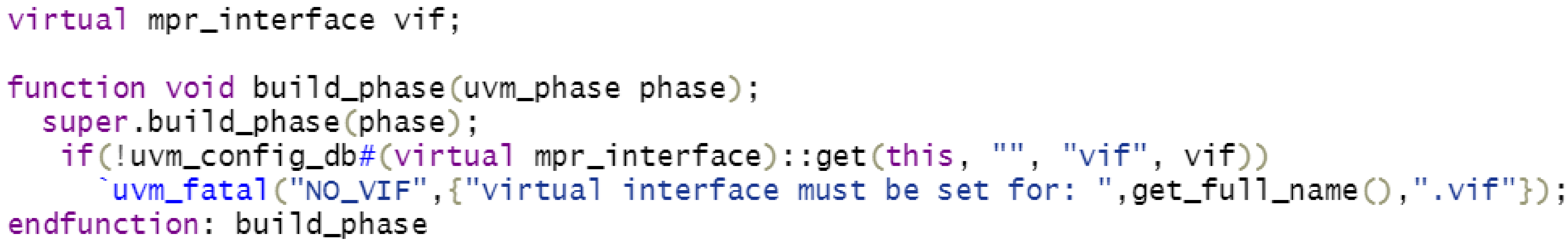
Step 1: Create a custom class inherited from uvm_driver, register with factory and call new



```
class mpr_driver extends uvm_driver #(mpr_seq_item);
  `uvm_component_utils(mpr_driver)

  function new (string name, uvm_component parent);
    super.new(name, parent);
  endfunction : new
  //Rest of the code
endclass
```

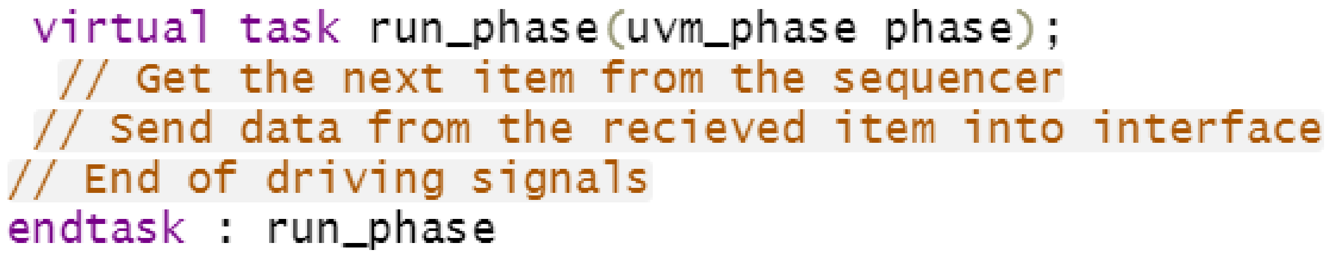
UVM Driver

Step 2: Declare a virtual interface handle and get them in build phase



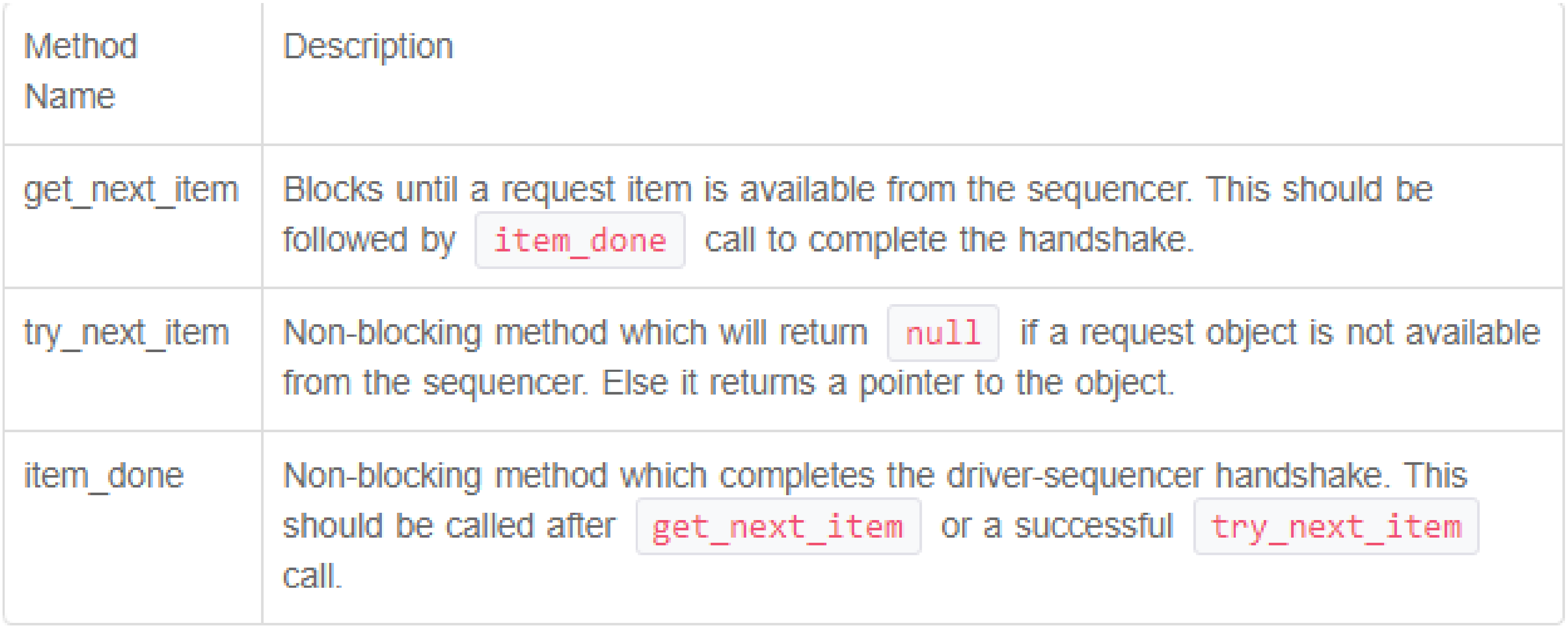
```
virtual mpr_interface vif;

function void build_phase(uvm_phase phase);
  super.build_phase(phase);
   if(!uvm_config_db#(virtual mpr_interface)::get(this, "", "vif", vif))
     `uvm_fatal("NO_VIF",{"virtual interface must be set for: ",get_full_name(),".vif"});
endfunction: build_phase
```

Step 3: Code the run phase



```
 virtual task run_phase(uvm_phase phase);
  // Get the next item from the sequencer
 // Send data from the recieved item into interface
// End of driving signals
endtask : run_phase
```

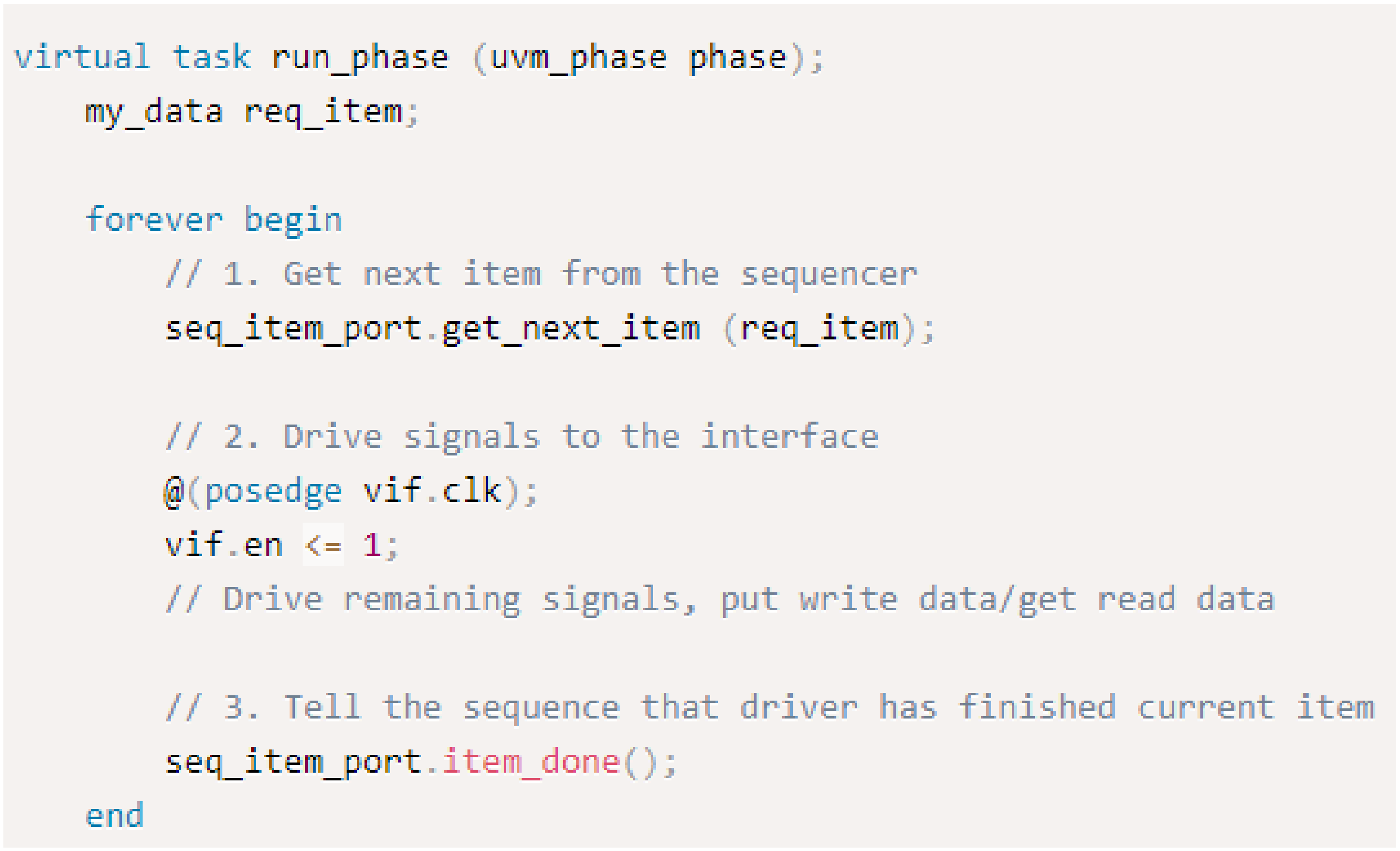
UVM Driver-Sequencer handshake



Method Name	Description
get_next_item	Blocks until a request item is available from the sequencer. This should be followed by `item_done` call to complete the handshake.
try_next_item	Non-blocking method which will return `null` if a request object is not available from the sequencer. Else it returns a pointer to the object.
item_done	Non-blocking method which completes the driver-sequencer handshake. This should be called after `get_next_item` or a successful `try_next_item` call.

Driver/Sequencer APIs

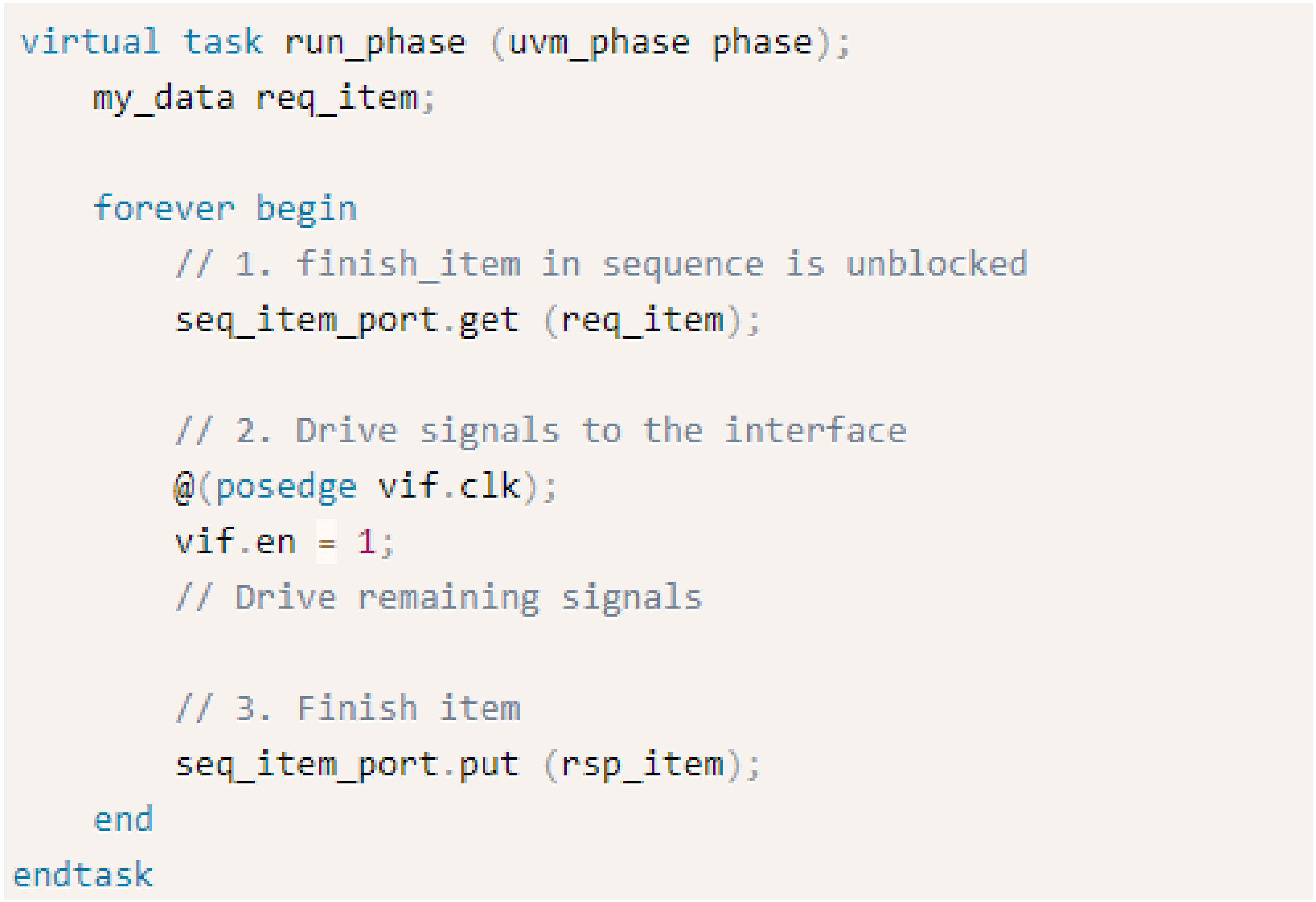
1. get_next_item followed by item_done



```
virtual task run_phase (uvm_phase phase);
    my_data req_item;

    forever begin
        // 1. Get next item from the sequencer
        seq_item_port.get_next_item (req_item);

        // 2. Drive signals to the interface
        @(posedge vif.clk);
        vif.en <= 1;
        // Drive remaining signals, put write data/get read data

        // 3. Tell the sequence that driver has finished current item
        seq_item_port.item_done();
    end
```

Driver/Sequencer APIs

2. get followed by put

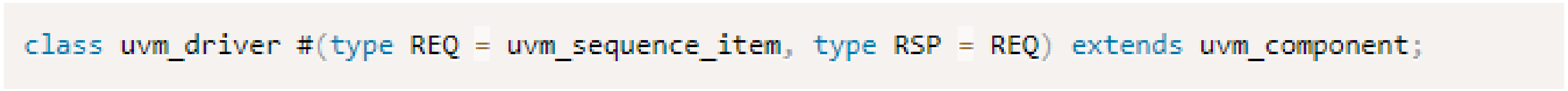


```
virtual task run_phase (uvm_phase phase);
    my_data req_item;

    forever begin
        // 1. finish_item in sequence is unblocked
        seq_item_port.get (req_item);

        // 2. Drive signals to the interface
        @(posedge vif.clk);
        vif.en = 1;
        // Drive remaining signals

        // 3. Finish item
        seq_item_port.put (rsp_item);
    end
endtask
```

```
class my_driver extends uvm_driver #(my_data);
  `uvm_component_utils (my_driver)

   virtual  dut_if   vif;

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

   virtual function void build_phase (uvm_phase phase);
      super.build_phase (phase);
      if (! uvm_config_db #(virtual dut_if) :: get (this, "", "vif", vif)) begin
         `uvm_fatal (get_type_name (), "Didn't get handle to virtual interface dut_if")
      end
   endfunction

   task run_phase (uvm_phase phase);
      my_data data_obj;
      super.run_phase (phase);

      forever begin
         `uvm_info (get_type_name (), $sformatf ("Waiting for data from sequencer"), UVM_MEDIUM)
         seq_item_port.get_next_item (data_obj);
         drive_item (data_obj);
         seq_item_port.item_done ();
      end
   endtask

   virtual task drive_item (my_data data_obj);
      // Drive based on bus protocol
   endtask
endclass
```

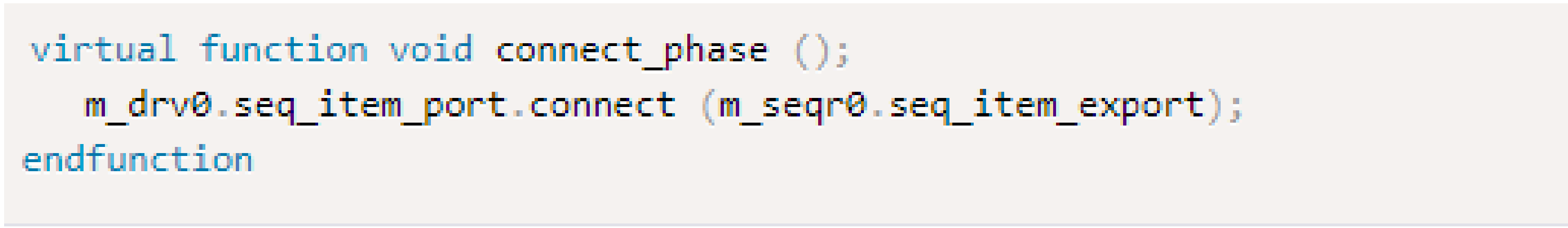
UVM Driver

- The base driver has a uvm_seq_item_pull_port through which it requests for new transactions from the export connected to it.



```
class uvm_driver #(type REQ = uvm_sequence_item, type RSP = REQ) extends uvm_component;
```

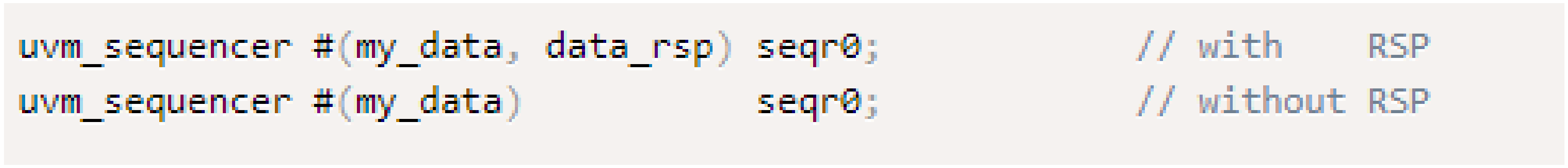
- The driver’s port and sequencer’s export are connected during the connect_phase() of an environment/agent class.



```
 virtual function void connect_phase ();
   m_drv0.seq_item_port.connect (m_seqr0.seq_item_export);
endfunction
```

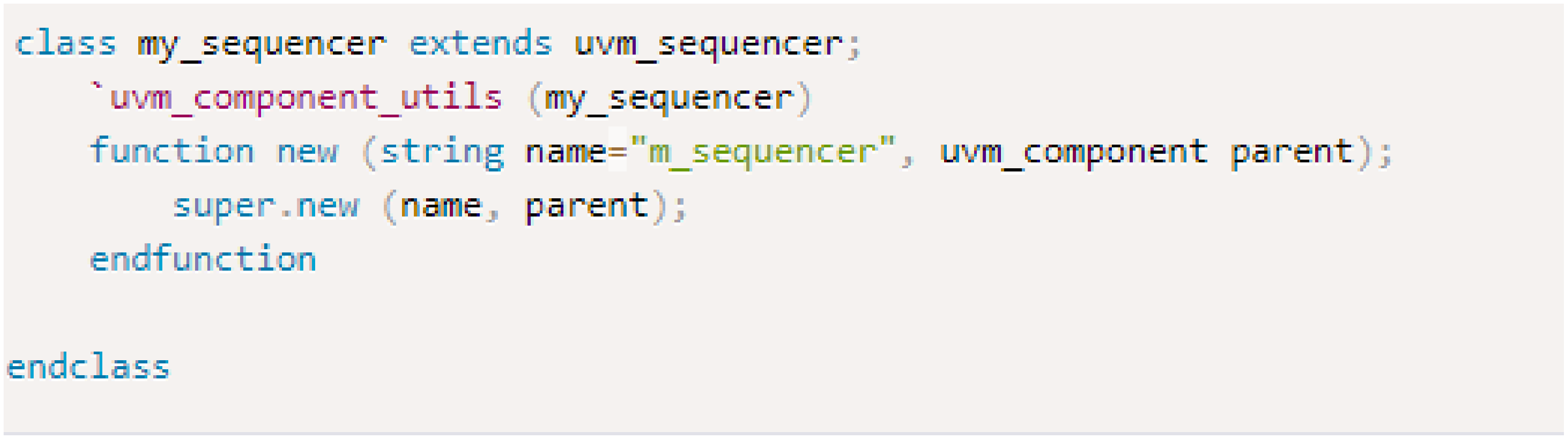
UVM Sequencer

- Generates data transactions as class objects and send it to the driver for execution



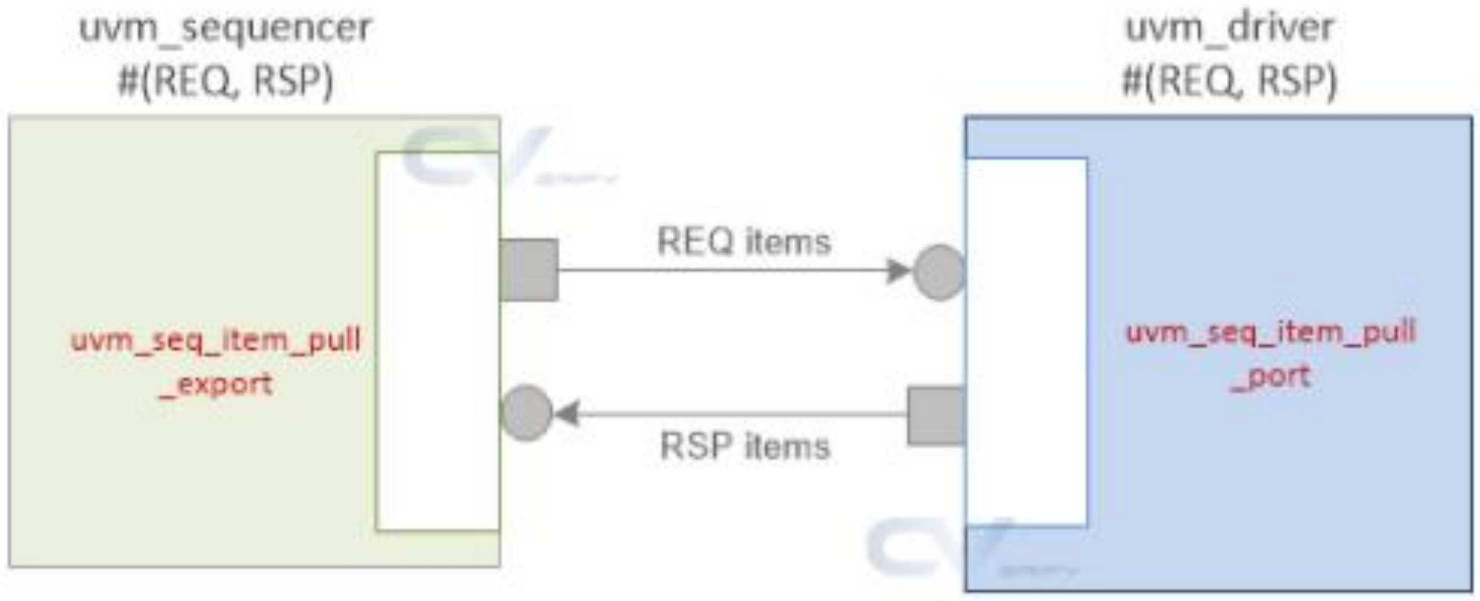
```
uvm_sequencer #(my_data, data_rsp) seqr0;               // with    RSP
uvm_sequencer #(my_data)           seqr0;               // without RSP
```

- Custom sequencer

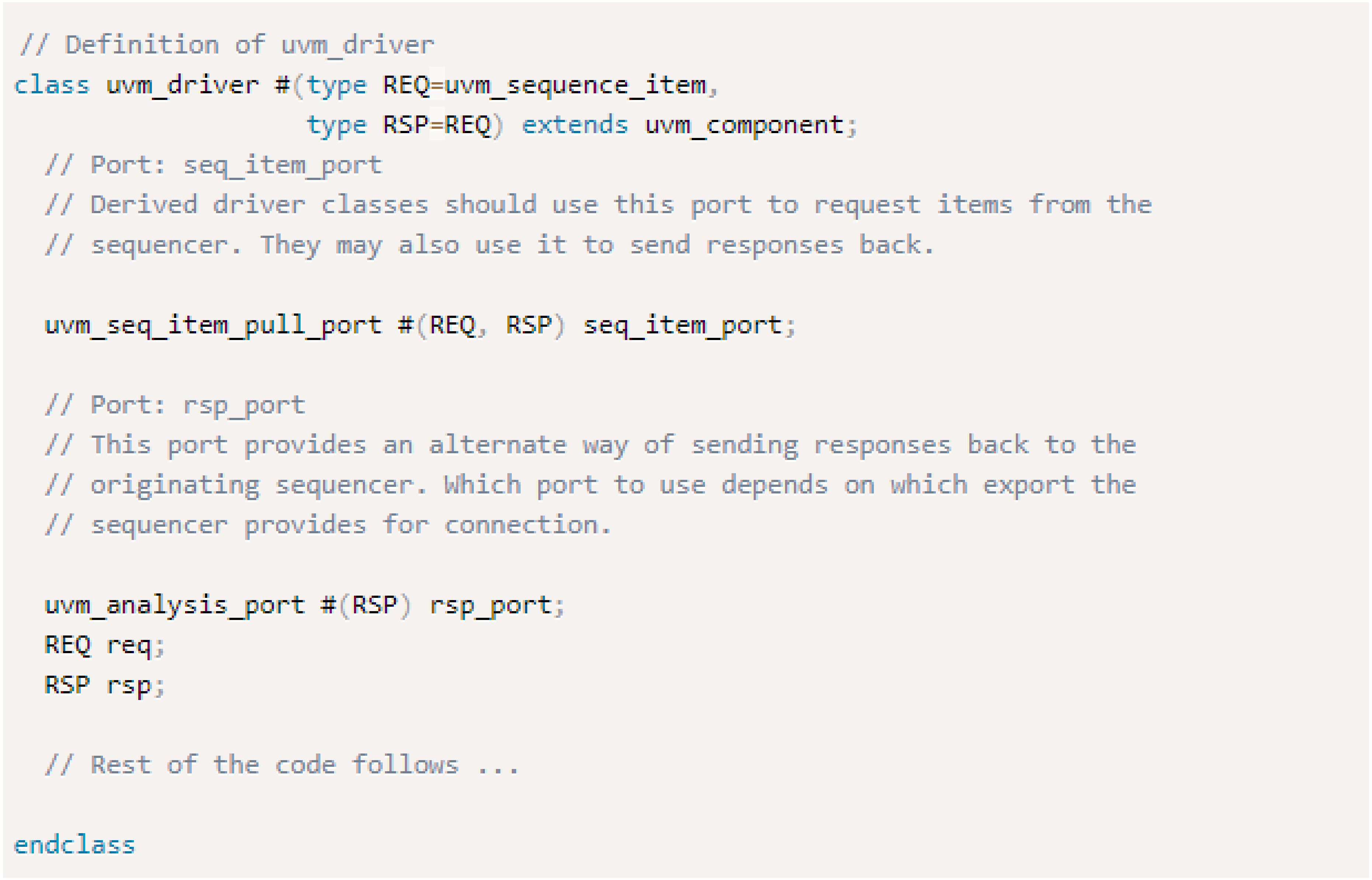


```
class my_sequencer extends uvm_sequencer;
    `uvm_component_utils (my_sequencer)
    function new (string name="m_sequencer", uvm_component parent);
        super.new (name, parent);
    endfunction

endclass
```

Driver-Sequencer Handshake



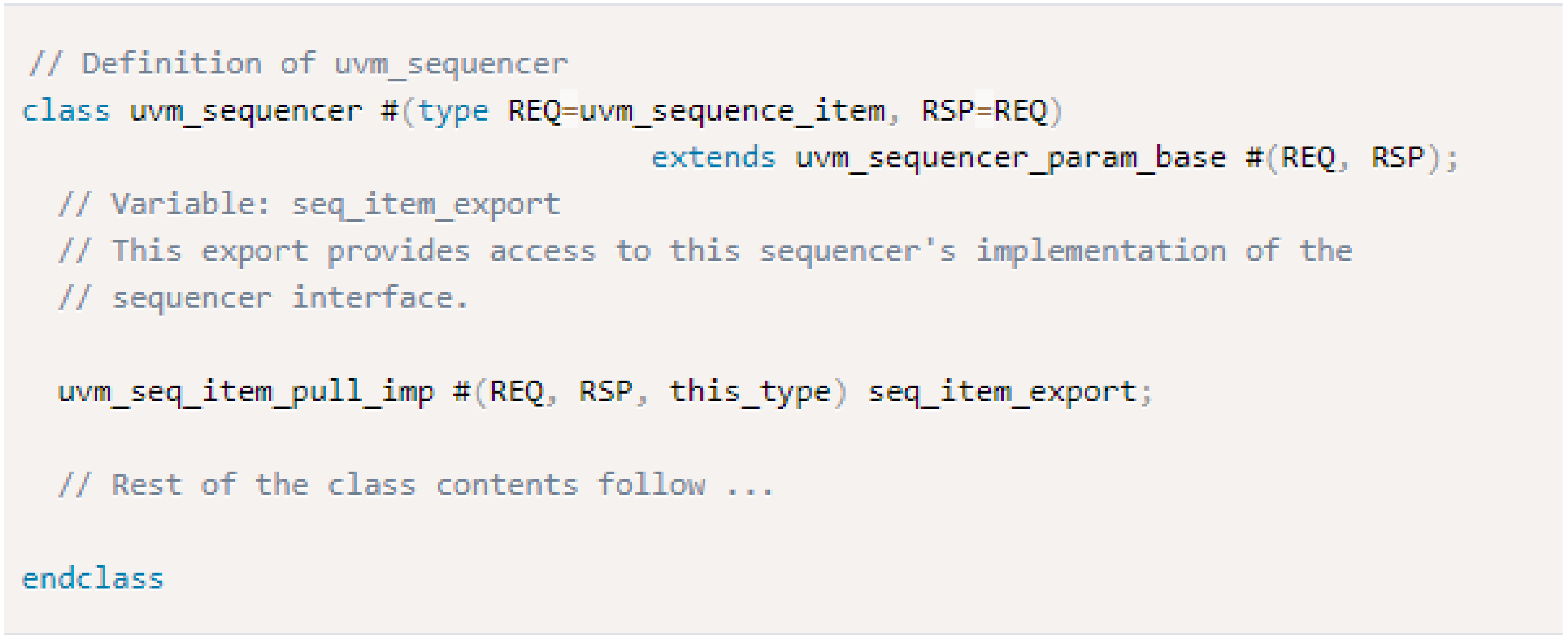
Driver port



```
// Definition of uvm_driver
class uvm_driver #(type REQ=uvm_sequence_item,
                   type RSP=REQ) extends uvm_component;
  // Port: seq_item_port
  // Derived driver classes should use this port to request items from the
  // sequencer. They may also use it to send responses back.

  uvm_seq_item_pull_port #(REQ, RSP) seq_item_port;

  // Port: rsp_port
  // This port provides an alternate way of sending responses back to the
  // originating sequencer. Which port to use depends on which export the
  // sequencer provides for connection.

  uvm_analysis_port #(RSP) rsp_port;
  REQ req;
  RSP rsp;

  // Rest of the code follows ...

endclass
```

Sequencer port

- A uvm_sequencer has an inbuilt TLM pull implementation port called seq_item_export that is used to connect with the driver’s pull port.

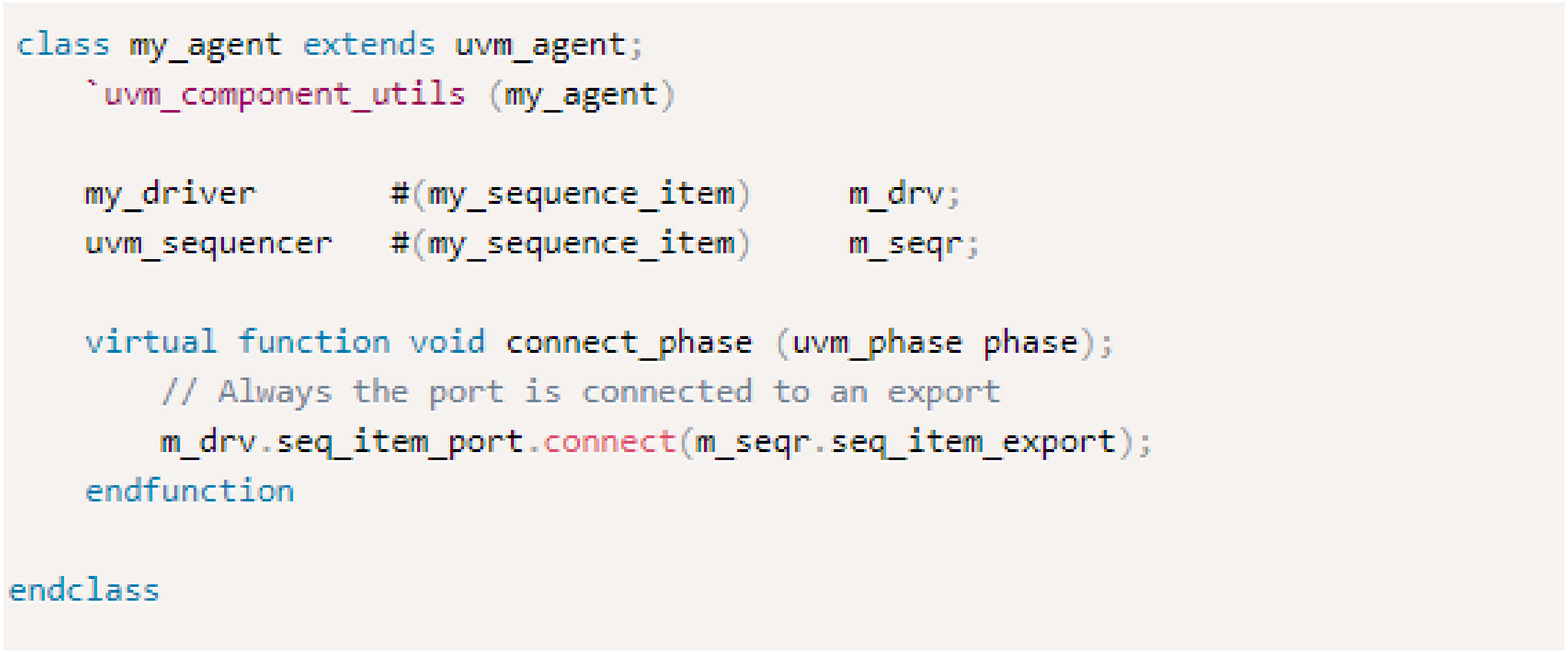


```
// Definition of uvm_sequencer
class uvm_sequencer #(type REQ=uvm_sequence_item, RSP=REQ)
                                  extends uvm_sequencer_param_base #(REQ, RSP);
  // Variable: seq_item_export
  // This export provides access to this sequencer's implementation of the
  // sequencer interface.

  uvm_seq_item_pull_imp #(REQ, RSP, this_type) seq_item_export;

  // Rest of the class contents follow ...

endclass
```

Connection of driver & sequencer

- The port in uvm_driver is connected to the export in uvm_sequencer in the connect phase of the UVM component in which both driver and sequencer are instantiated, typically in a uvm_agent.
- The connect between driver and sequencer is a one-to-one connection.
- Multiple drivers aren’t connected to a sequencer nor are multiple sequencers connected to a single driver.
- Once the connection is made, the driver can utilize the API calls in the TLM port definitions to receive sequence items from the sequencer.

Connection of driver & sequencer



```
class my_agent extends uvm_agent;
    `uvm_component_utils (my_agent)

    my_driver       #(my_sequence_item)     m_drv;
    uvm_sequencer   #(my_sequence_item)     m_seqr;

    virtual function void connect_phase (uvm_phase phase);
        // Always the port is connected to an export
        m_drv.seq_item_port.connect(m_seqr.seq_item_export);
    endfunction

endclass
```

UVM Monitor

- What is a monitor?
- A UVM component that captures signal activity from design interface and translates it into transaction level objects that is sent to other components.

- What does it require?
- A virtual interface handle to the actual interface that the monitor is monitoring.
- TLM Analysis port declarations to broadcast captured data to others.

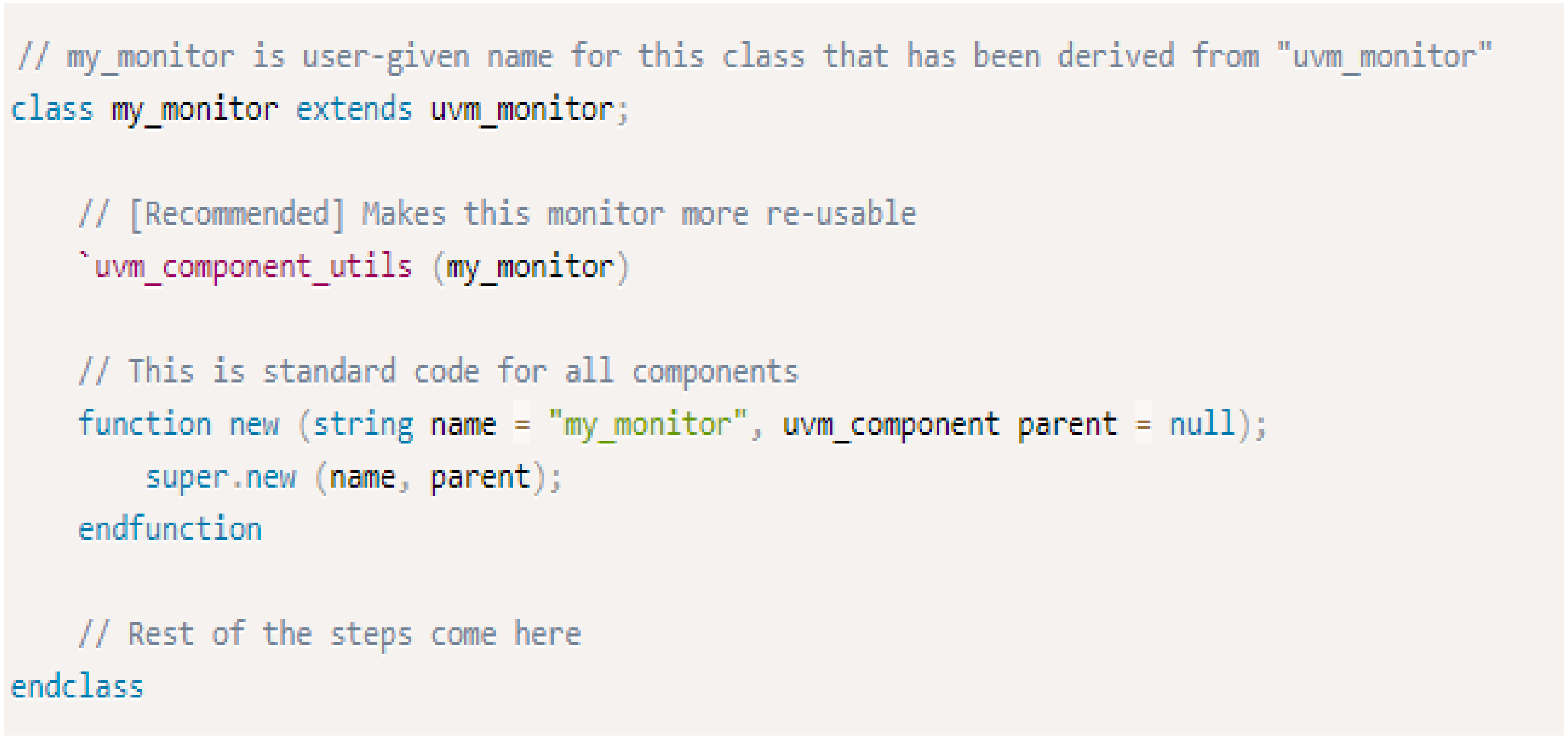
UVM Monitor

What does UVM Monitor do?

1. Collect bus or signal information through a virtual interface.
2. Collected data can be used for protocol checking and coverage (enable/disable with knobs).
3. Collected data is exported via an analysis port.

Step-1 to create UVM monitor

- Create custom class inherited from uvm_monitor, register with factory and call new.



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

    // [Recommended] Makes this monitor more re-usable
    `uvm_component_utils (my_monitor)

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

    // Rest of the steps come here
endclass
```

Step-2 to create UVM monitor

- Declare analysis ports and virtual interface handles

```
// Actual interface object is later obtained by doing a get() call on uvm_config_db
virtual if_name vif;

// my_data is a custom class object used to encapsulate signal information
// and can be sent to other components
uvm_analysis_port  #(my_data) mon_analysis_port;
```

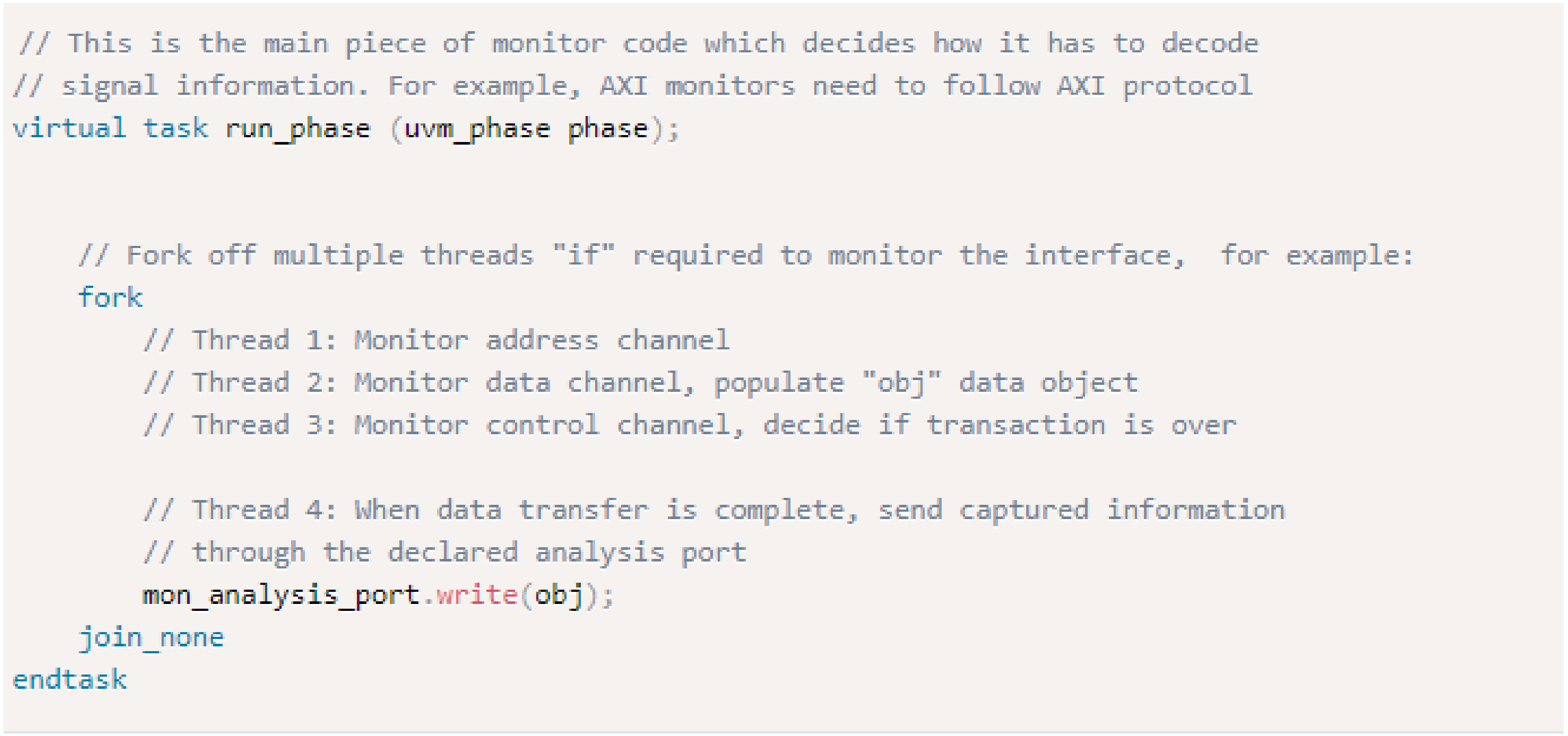
Step-3 to create UVM monitor

- Build the UVM monitor

```
virtual function void build_phase (uvm_phase phase);
   super.build_phase (phase);

   // Create an instance of the declared analysis port
   mon_analysis_port = new ("mon_analysis_port", this);

   // Get virtual interface handle from the configuration DB
   if (! uvm_config_db #(virtual if_name) :: get (this, "", "vif", vif)) begin
      `uvm_error (get_type_name (), "DUT interface not found")
   end
endfunction
```

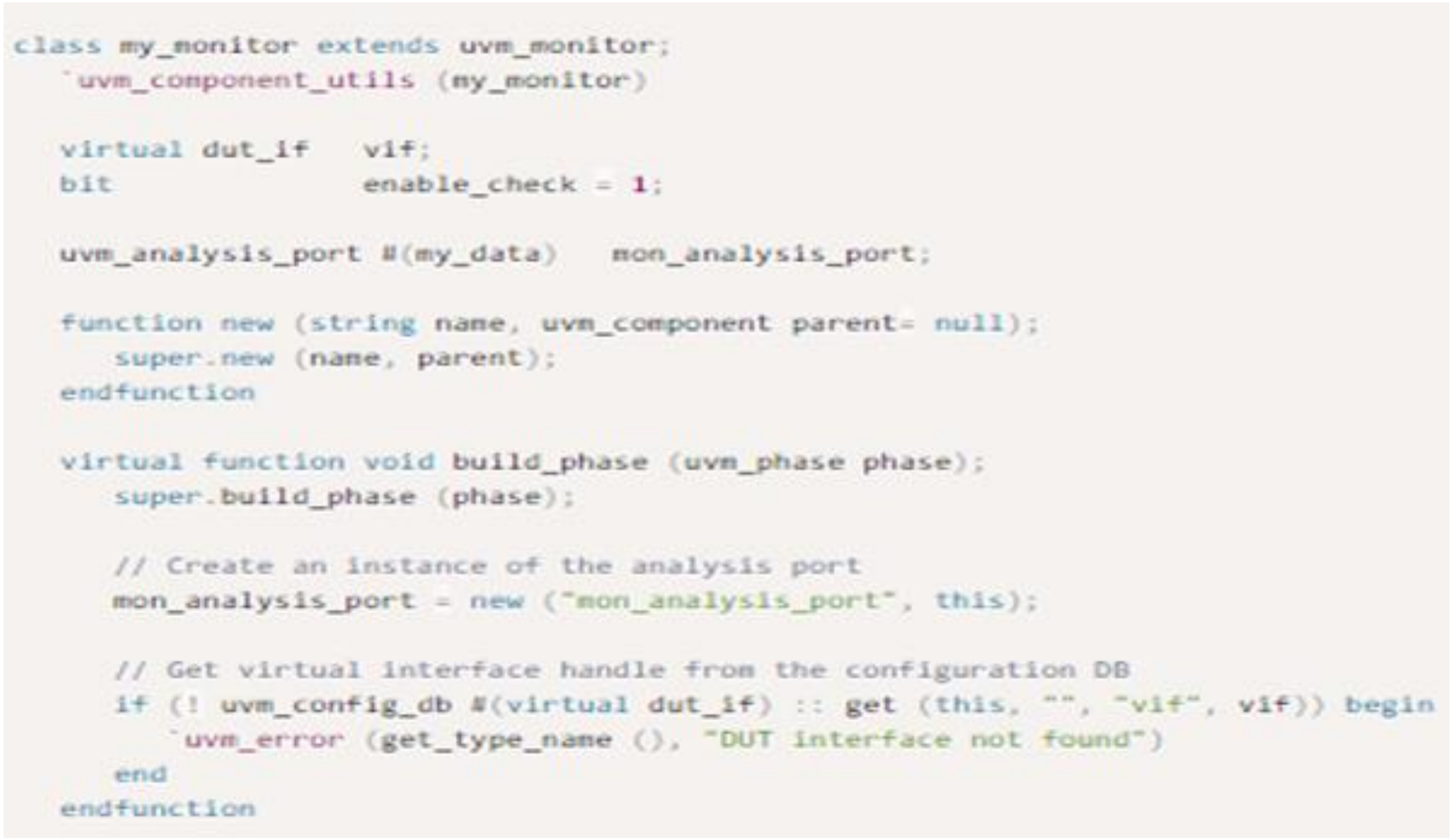
Step-4 to create UVM monitor

- Code the run_phase



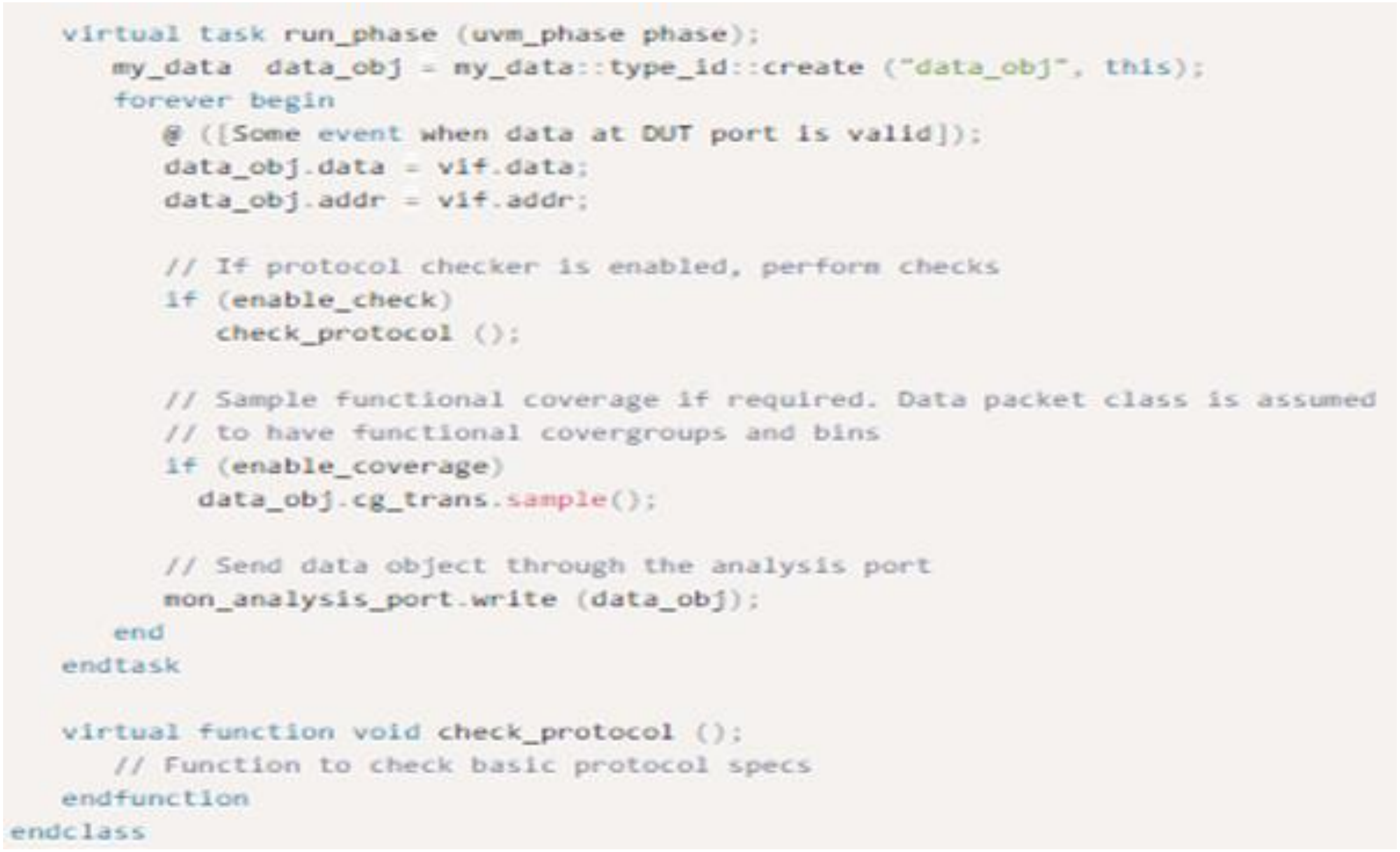
```
// This is the main piece of monitor code which decides how it has to decode
// signal information. For example, AXI monitors need to follow AXI protocol
virtual task run_phase (uvm_phase phase);


    // Fork off multiple threads "if" required to monitor the interface,  for example:
    fork
        // Thread 1: Monitor address channel
        // Thread 2: Monitor data channel, populate "obj" data object
        // Thread 3: Monitor control channel, decide if transaction is over

        // Thread 4: When data transfer is complete, send captured information
        // through the declared analysis port
        mon_analysis_port.write(obj);
    join_none
endtask
```

Step-4 to create UVM monitor



```
class my_monitor extends uvm_monitor;
   `uvm_component_utils (my_monitor)

   virtual dut_if   vif;
   bit              enable_check = 1;

   uvm_analysis_port #(my_data)   mon_analysis_port;

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

   virtual function void build_phase (uvm_phase phase);
      super.build_phase (phase);

      // Create an instance of the analysis port
      mon_analysis_port = new ("mon_analysis_port", this);

      // Get virtual interface handle from the configuration DB
      if (! uvm_config_db #(virtual dut_if) :: get (this, "", "vif", vif)) begin
         `uvm_error (get_type_name (), "DUT interface not found")
      end
   endfunction
```

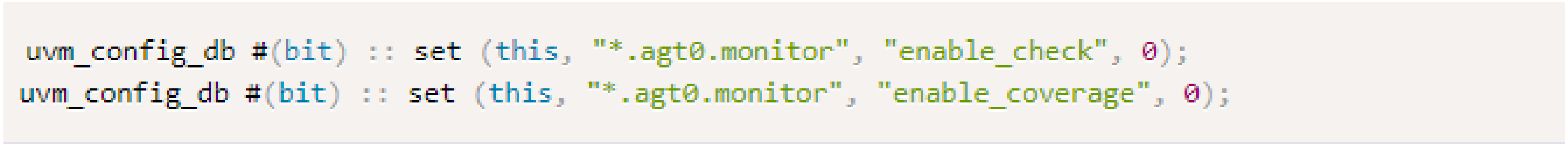
Step-4 to create UVM monitor



```
   virtual task run_phase (uvm_phase phase);
      my_data  data_obj = my_data::type_id::create ("data_obj", this);
      forever begin
         @ ([Some event when data at DUT port is valid]);
         data_obj.data = vif.data;
         data_obj.addr = vif.addr;

         // If protocol checker is enabled, perform checks
         if (enable_check)
            check_protocol ();

         // Sample functional coverage if required. Data packet class is assumed
         // to have functional covergroups and bins
         if (enable_coverage)
           data_obj.cg_trans.sample();

         // Send data object through the analysis port
         mon_analysis_port.write (data_obj);
      end
   endtask

   virtual function void check_protocol ();
      // Function to check basic protocol specs
   endfunction
endclass
```

Knobs in UVM Monitor

The knobs can be disabled from the test by using the UVM database.



```
 uvm_config_db #(bit) :: set (this, "*.agt0.monitor", "enable_check", 0);
uvm_config_db #(bit) :: set (this, "*.agt0.monitor", "enable_coverage", 0);
```

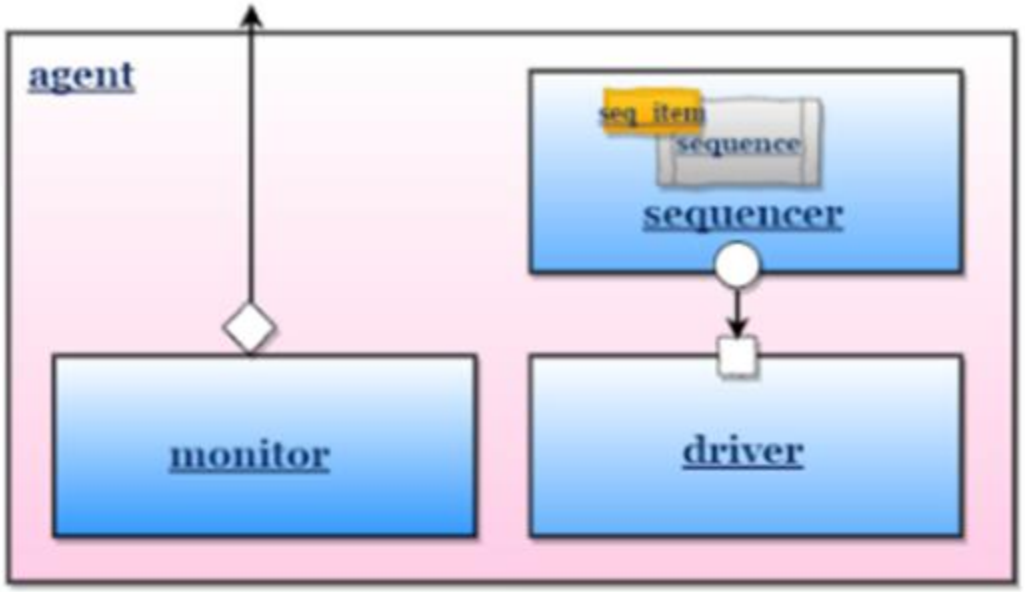
UVM Agent

Active

- Instantiates all three components [Sequencer, Driver, Monitor]
- Enables data to be driven to DUT via driver

Passive

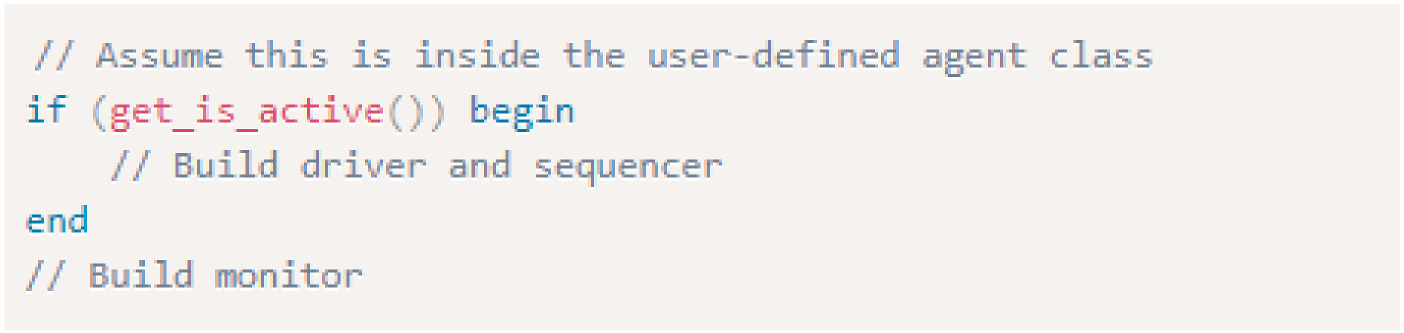
- Only instantiate the monitor
- Used for checking and coverage only
- Useful when there's no data item to be driven to DUT



```
set_config_int("path_to_agent", "is_active", UVM_ACTIVE);
set_config_int("path_to_agent", "is_active", UVM_PASSIVE);
```

UVM Agent – Active or Passive?

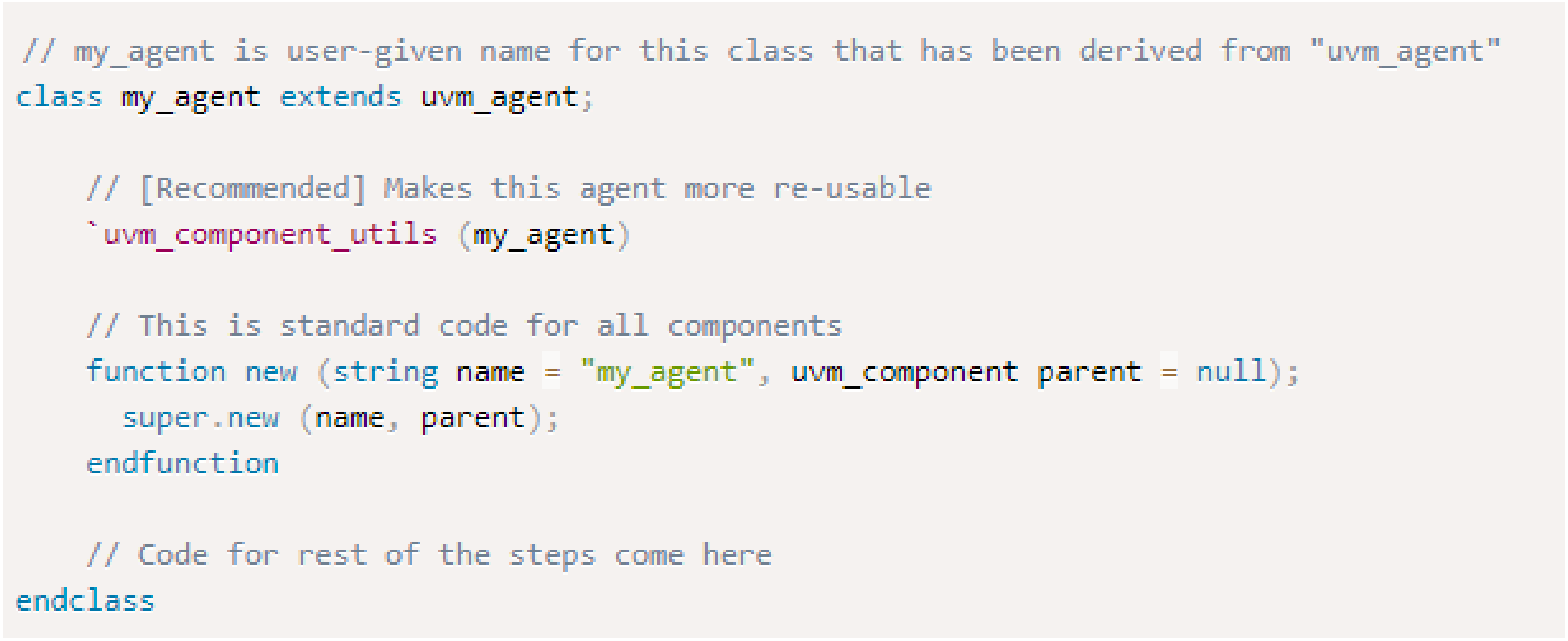
- get_is_active() – returns the state of the requested UVM agent



```
// Assume this is inside the user-defined agent class
if (get_is_active()) begin
    // Build driver and sequencer
end
// Build monitor
```

Creating a UVM Agent

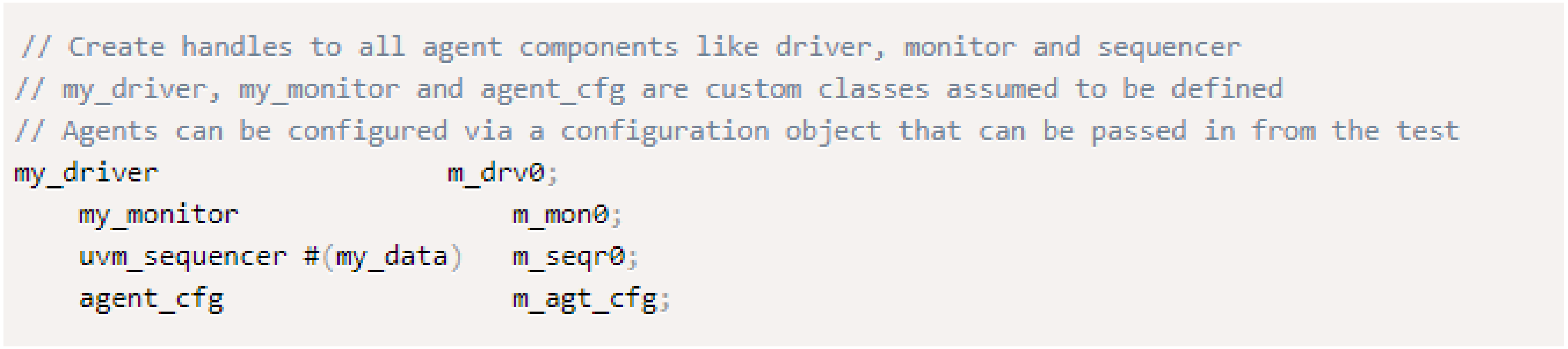
1. Create a custom class inherited from uvm_agent, register with factory and call new.



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

    // [Recommended] Makes this agent more re-usable
    `uvm_component_utils (my_agent)

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

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

Creating a UVM Agent

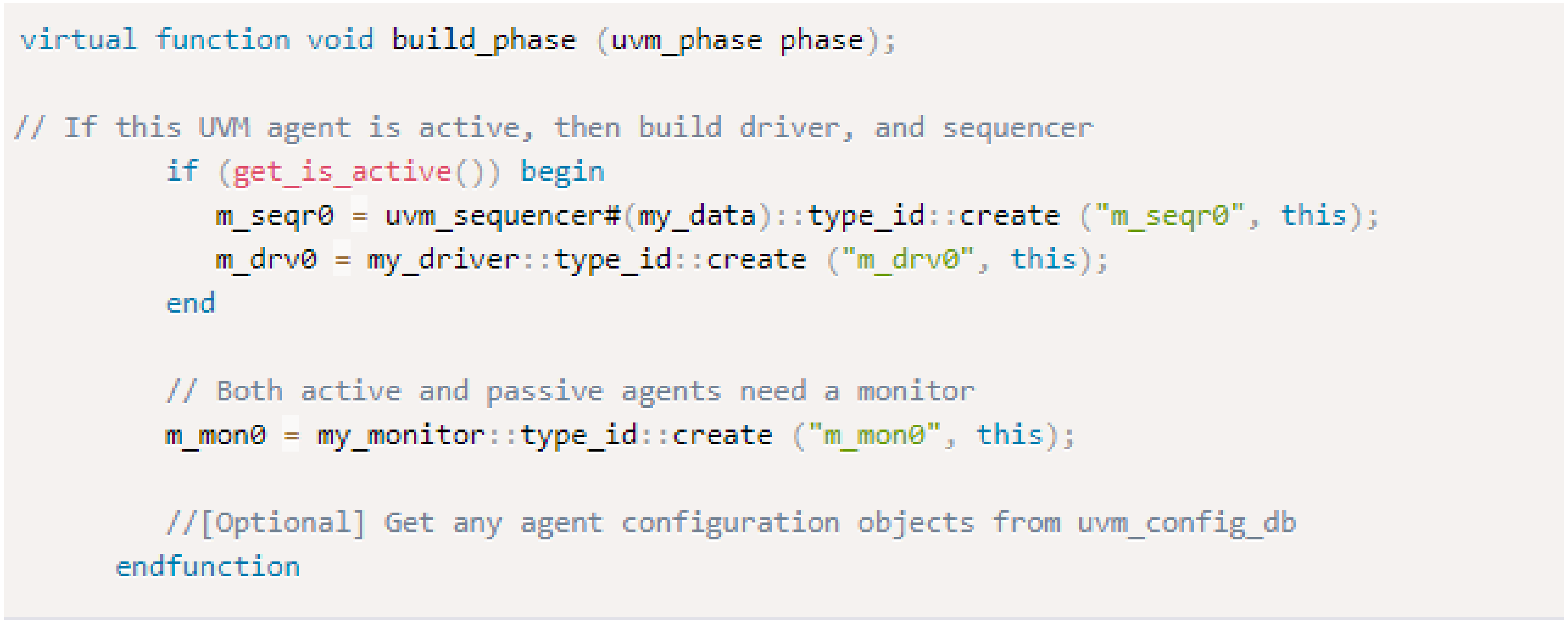
2. Instantiate agent components



```
// Create handles to all agent components like driver, monitor and sequencer
// my_driver, my_monitor and agent_cfg are custom classes assumed to be defined
// Agents can be configured via a configuration object that can be passed in from the test
my_driver                  m_drv0;
    my_monitor                 m_mon0;
    uvm_sequencer #(my_data)   m_seqr0;
    agent_cfg                  m_agt_cfg;
```

Creating a UVM Agent

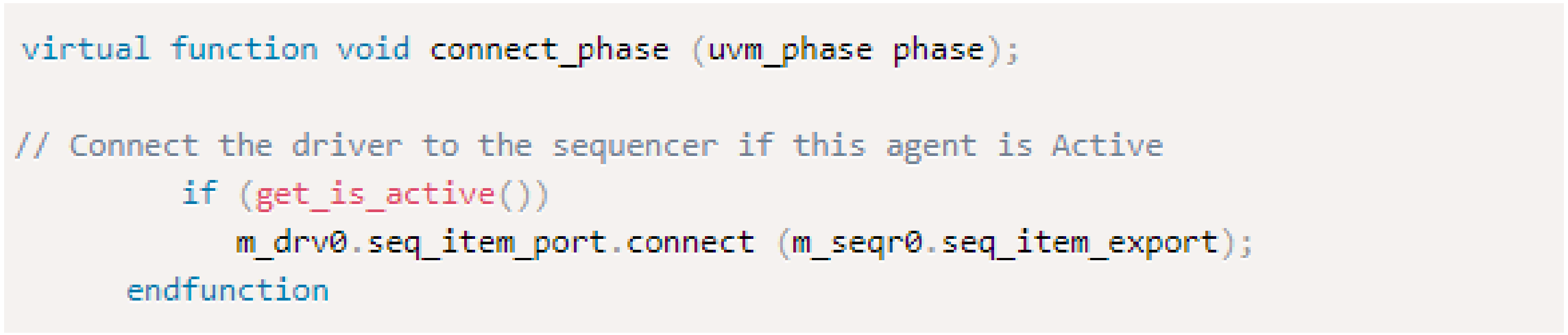
3. Instantiate and build components



```
virtual function void build_phase (uvm_phase phase);

// If this UVM agent is active, then build driver, and sequencer
         if (get_is_active()) begin
            m_seqr0 = uvm_sequencer#(my_data)::type_id::create ("m_seqr0", this);
            m_drv0 = my_driver::type_id::create ("m_drv0", this);
         end

         // Both active and passive agents need a monitor
         m_mon0 = my_monitor::type_id::create ("m_mon0", this);

         //[Optional] Get any agent configuration objects from uvm_config_db
      endfunction
```

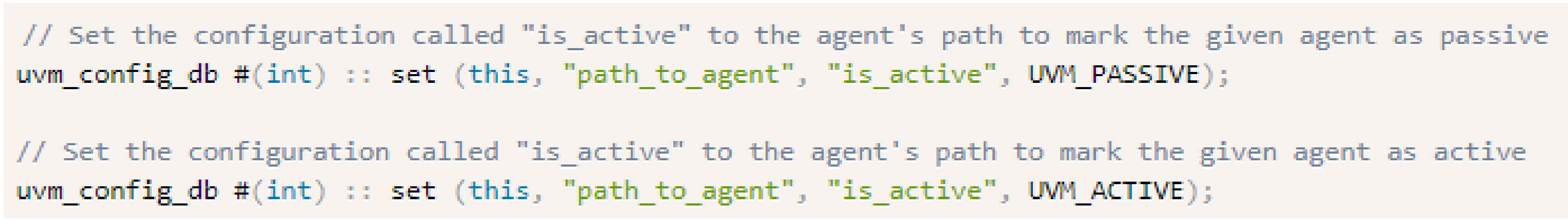
Creating a UVM Agent

4. Connect agent component together



```
virtual function void connect_phase (uvm_phase phase);

// Connect the driver to the sequencer if this agent is Active
         if (get_is_active())
            m_drv0.seq_item_port.connect (m_seqr0.seq_item_export);
      endfunction
```

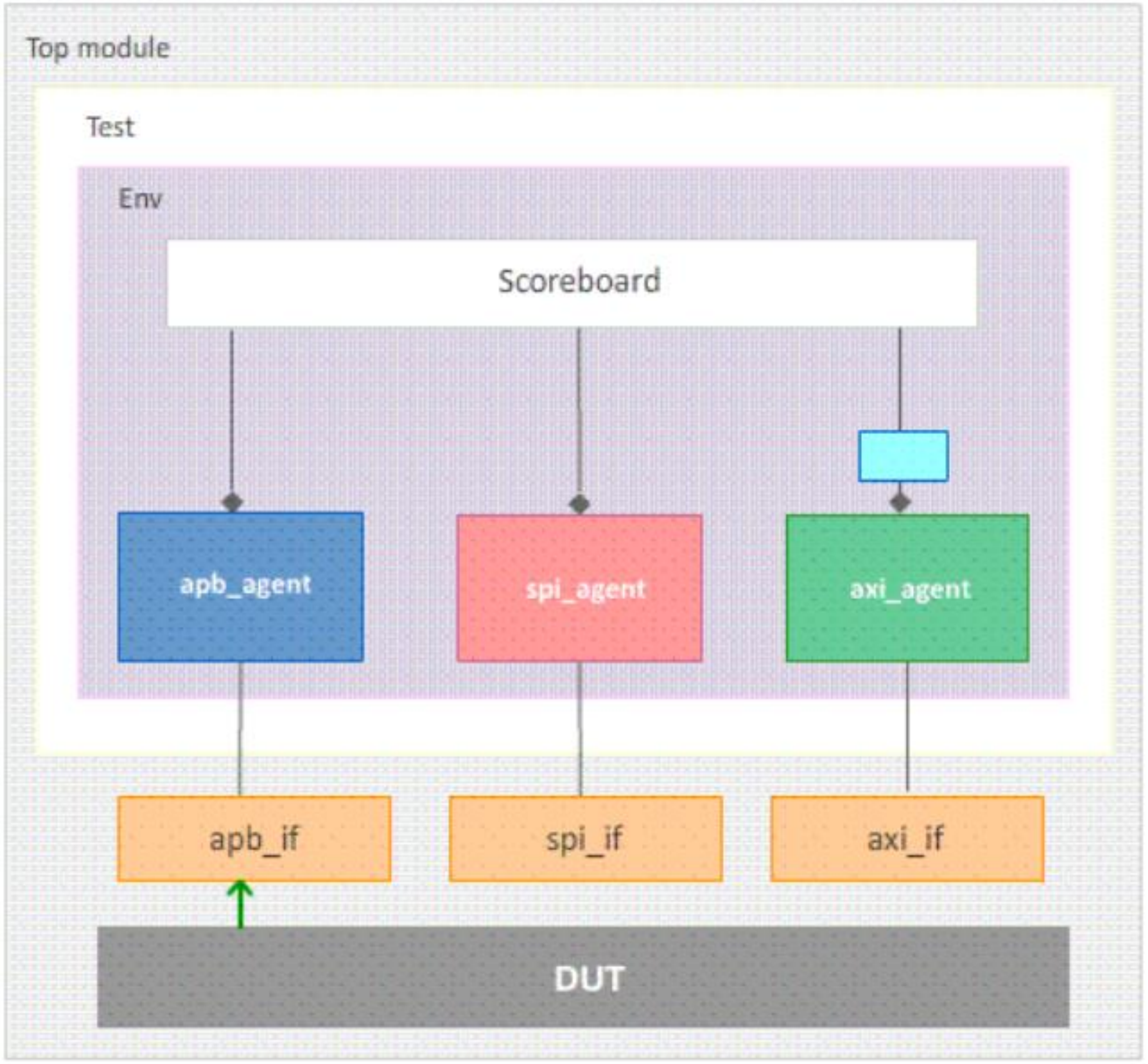
- How to configure UVM agent as active or passive?



```
// Set the configuration called "is_active" to the agent's path to mark the given agent as passive
uvm_config_db #(int) :: set (this, "path_to_agent", "is_active", UVM_PASSIVE);

// Set the configuration called "is_active" to the agent's path to mark the given agent as active
uvm_config_db #(int) :: set (this, "path_to_agent", "is_active", UVM_ACTIVE);
```

What does UVM agent do?

- Provides protocol specific tasks to generate transactions, check the results and perform coverage.



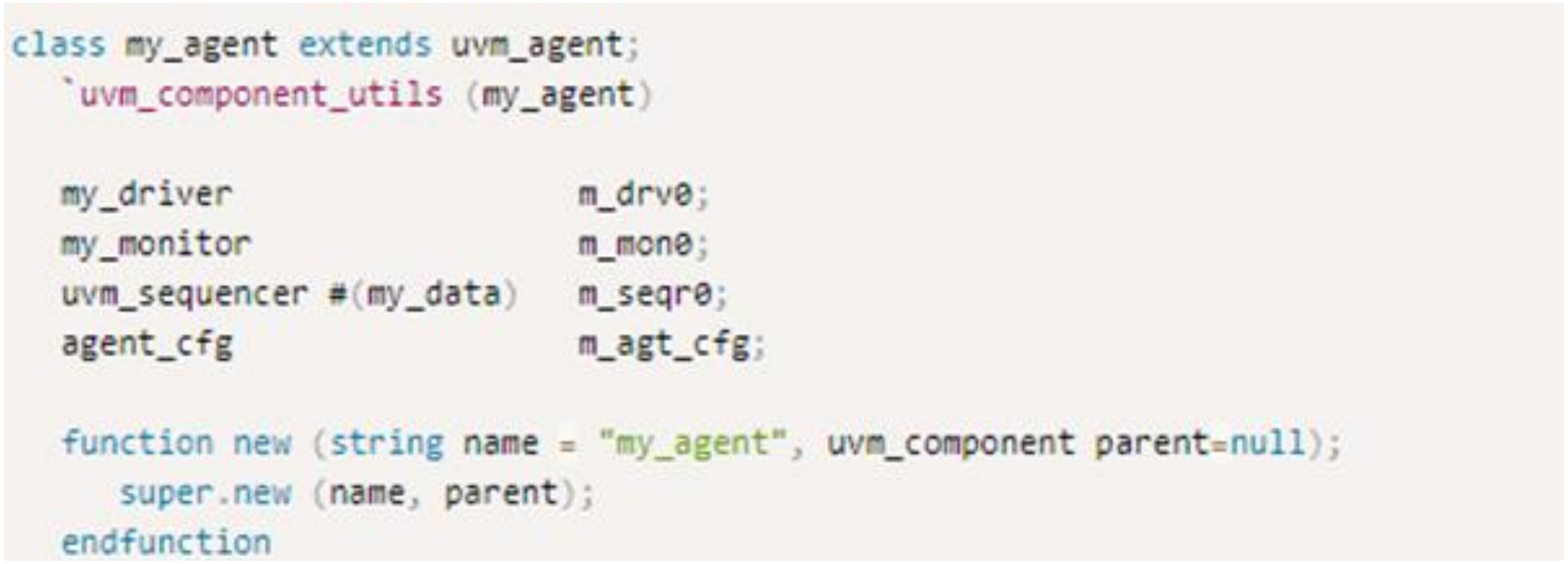
Multiple agents can be plugged into the environment

Each agent can drive protocol transactions to DUT via a specific interface

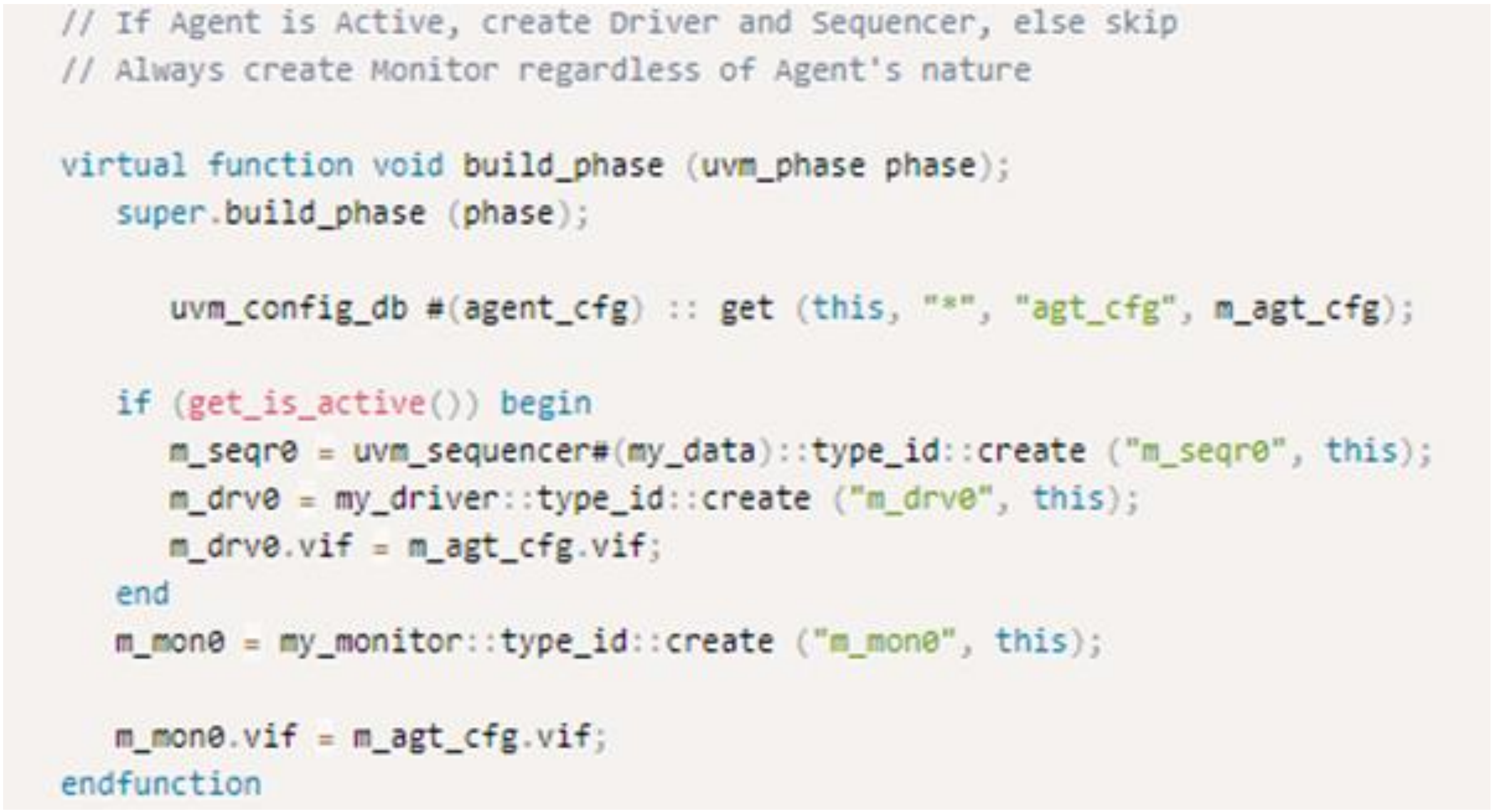
Each agent has a Sequencer, Driver and Monitor

An agent can be made either passive or active

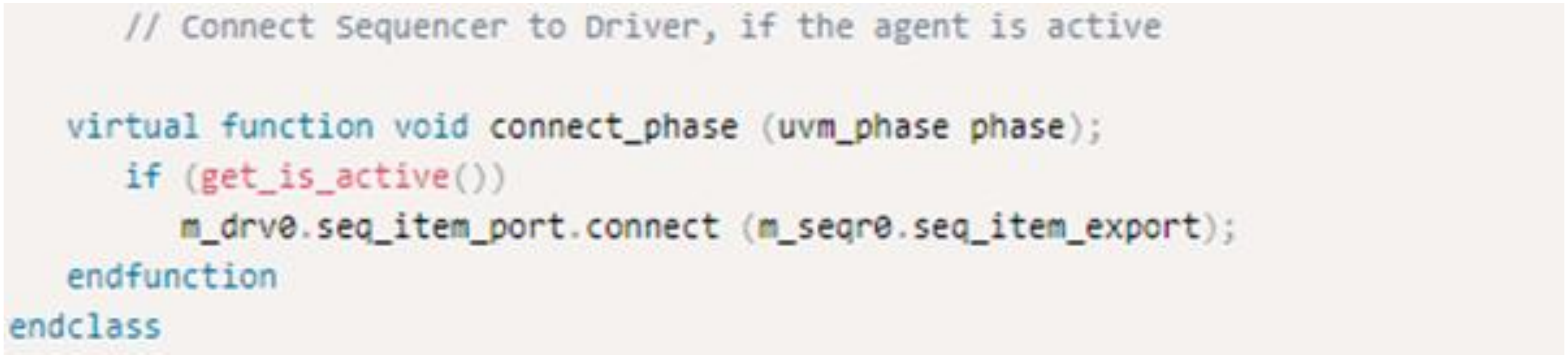
Agents send data to scoreboard via an analysis port



```
class my_agent extends uvm_agent;
  `uvm_component_utils (my_agent)

  my_driver                     m_drv0;
  my_monitor                    m_mon0;
  uvm_sequencer #(my_data)      m_seqr0;
  agent_cfg                     m_agt_cfg;

  function new (string name = "my_agent", uvm_component parent=null);
    super.new (name, parent);
  endfunction
```



```
// If Agent is Active, create Driver and Sequencer, else skip
// Always create Monitor regardless of Agent's nature

virtual function void build_phase (uvm_phase phase);
   super.build_phase (phase);

      uvm_config_db #(agent_cfg) :: get (this, "*", "agt_cfg", m_agt_cfg);

   if (get_is_active()) begin
      m_seqr0 = uvm_sequencer#(my_data)::type_id::create ("m_seqr0", this);
      m_drv0 = my_driver::type_id::create ("m_drv0", this);
      m_drv0.vif = m_agt_cfg.vif;
   end
   m_mon0 = my_monitor::type_id::create ("m_mon0", this);

   m_mon0.vif = m_agt_cfg.vif;
endfunction
```



```
    // Connect Sequencer to Driver, if the agent is active

  virtual function void connect_phase (uvm_phase phase);
    if (get_is_active())
      m_drv0.seq_item_port.connect (m_seqr0.seq_item_export);
  endfunction
endclass
```