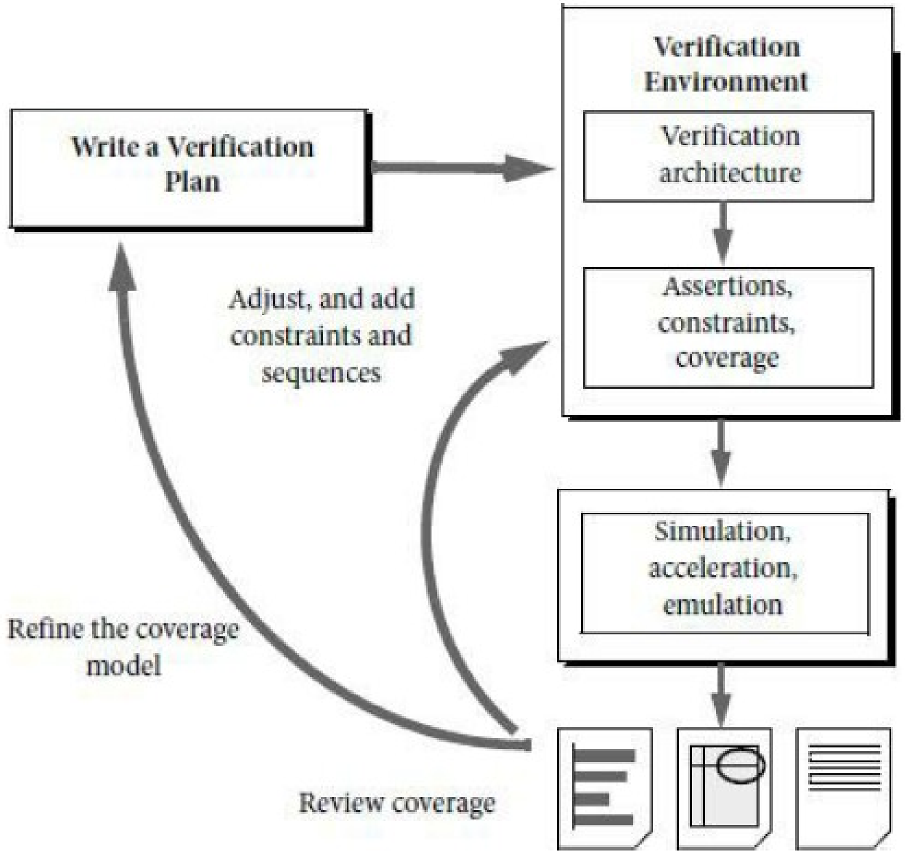
Universal Verification Methodology

Pankaj Badhe

UVM

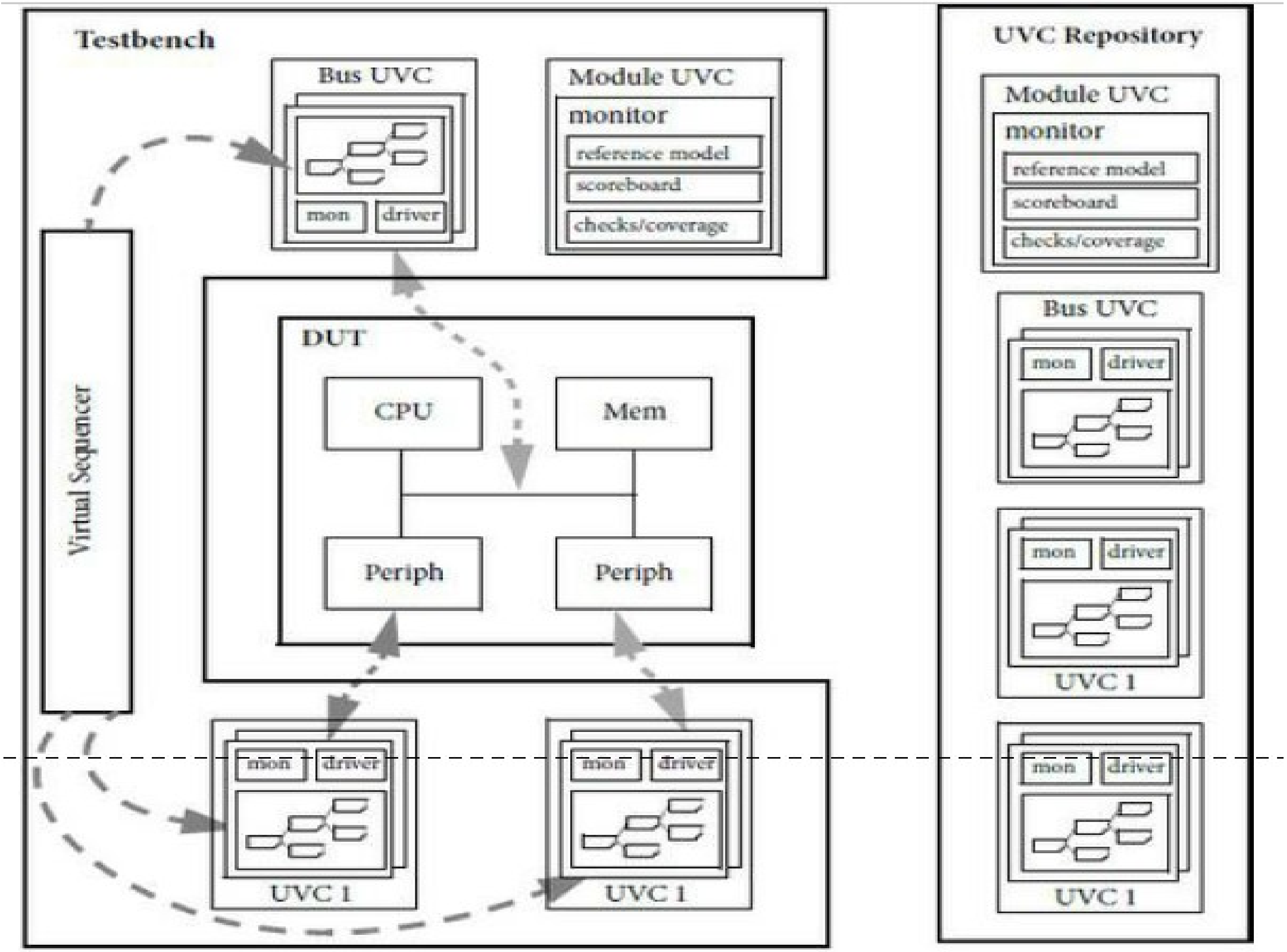

Write a Verification Plan
Verification Environment
Verification architecture
Assertions, constraints, coverage
Adjust, and add constraints and sequences
Simulation, acceleration, emulation
Refine the coverage model
Review coverage

UVM Testbench and Environments

- composed of reusable UVM-compliant universal verification components (UVCs).
- UVC is an encapsulated, ready-to-use and configurable verification environment intended for an interface protocol, a design sub-module, or even for software verification.
- Each UVC follows a consistent architecture and contains a complete set of elements for sending stimulus, as well as checking and collecting coverage information for a specific protocol or design.

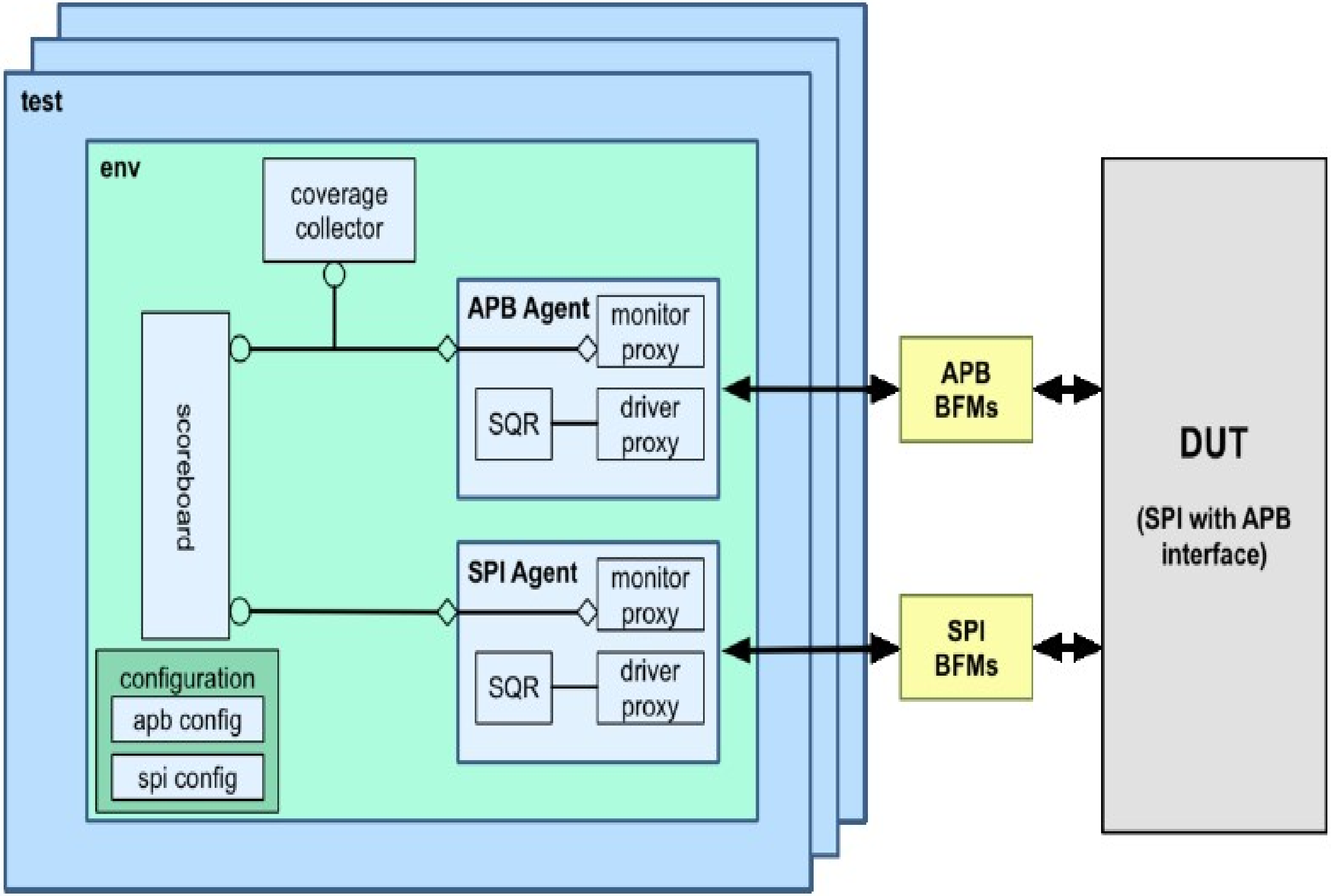
Agents

- Sequencers, drivers, monitors, and collectors can be reused independently, but this requires the environment integrator to learn the names, roles, configuration, and hookup of each of these entities.
- To reduce the amount of work and knowledge required by the test writer, UVM recommends that environment developers create a more abstract container called an agent.
- Agents can emulate and verify DUT devices. They encapsulate a driver, sequencer, monitor, and collector (when applicable).
- UVCs can contain more than one agent. Some agents are proactive (for example, master or transmit agents) and initiate transactions to the DUT, while other agents (slave or receive agents) react to transaction requests. Agents should be configurable so that they can be either active or passive.
- Active agents emulate devices and drive transactions according to test directives. Passive agents only monitor DUT activity.


Testbench
Bus UVC
mon
driver
Module UVC
monitor
reference model
scoreboard
checks/coverage
Virtual Sequencer
DUT
CPU
Mem
Periph
Periph
mon
driver
UVC 1
mon
driver
UVC 1
UVC Repository
Module UVC
monitor
reference model
scoreboard
checks/coverage
Bus UVC
mon
driver
mon
driver
UVC 1
mon
driver
UVC 1

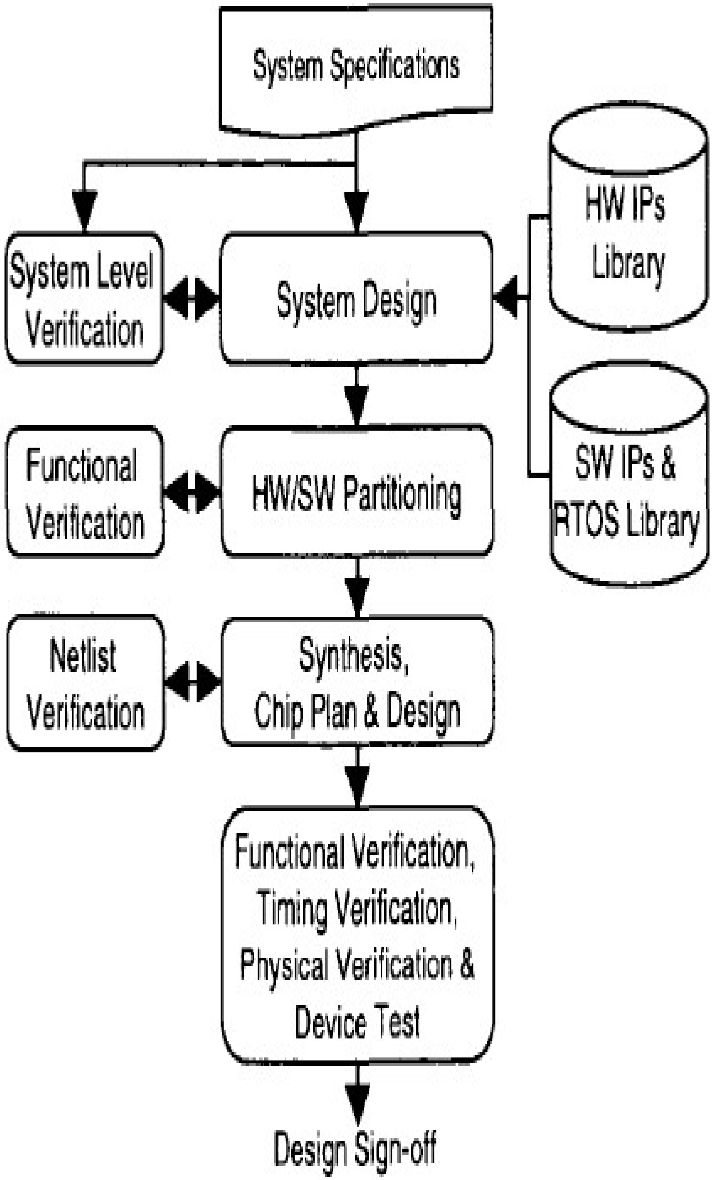
Environments

- □ The environment (env) is the top-level component of the UVC. It contains one or more agents, as well as other components such as a bus monitor.
- □ The env contains configuration properties that enable you to customize the topology and behavior to make it reusable. For example, active agents can be changed into passive agents when the verification environment is reused for system verification.
- □ The environment class (uvm_env) is designed to provide a flexible, reusable, and extendable verification component.
- □ The main function of the environment class is to model behavior by generating constrained-random traffic, monitoring DUT responses, checking the validity of the protocol activity, and collecting coverage.


test
env
coverage collector
scoreboard
APB Agent
monitor proxy
SQR
driver proxy
SPI Agent
monitor proxy
SQR
driver proxy
configuration
apb config
spi config
APB BFMs
SPI BFMs
DUT
(SPI with APB interface)

UVM Testbench Basi

- □ UVM employs a layered, object-oriented approach to testbench development that allows "separation of concerns" among the various team members.
- □ Each component in a UVM testbench has a specific purpose and a well-defined interface to the rest of the testbench to enhance productivity and facilitate reuse.
- □ When these components are assembled into a testbench, the result is a modular reusable verification environment that allows the test writer to think at the transaction level, focusing on the functionality that must be verified, while the testbench architect focuses on how the test interacts with the Design Under Test (DUT).



- The Design Under Test (DUT) is connected to a layer of transactors (drivers, monitors, responders).
- These transactors communicate with the DUT at the pin level by driving and sampling DUT signals, and with the rest of the UVM testbench by passing transaction objects.
- They convert data between pins and transactions, i.e. from/to signal to/from transaction level.
- The testbench layer above the transactor layer consists of components that interact exclusively at the transaction level, such as scoreboards, coverage collectors, stimulus generators, etc.
- All structural elements in a UVM testbench are extended from the uvm_component base class.

- The lowest level of a UVM testbench is interface-specific.
- For each interface, the UVM provides a uvm_agent that includes the driver, monitor, stimulus generator (sequencer) and (optionally) a coverage collector.
- The Agent thus embodies all of the protocol-specific communication with the DUT.
- The Agent(s) and other design-specific components are encapsulated in a uvm_env Environment component which is in turn instantiated and customize by a top-level uvm_test component.

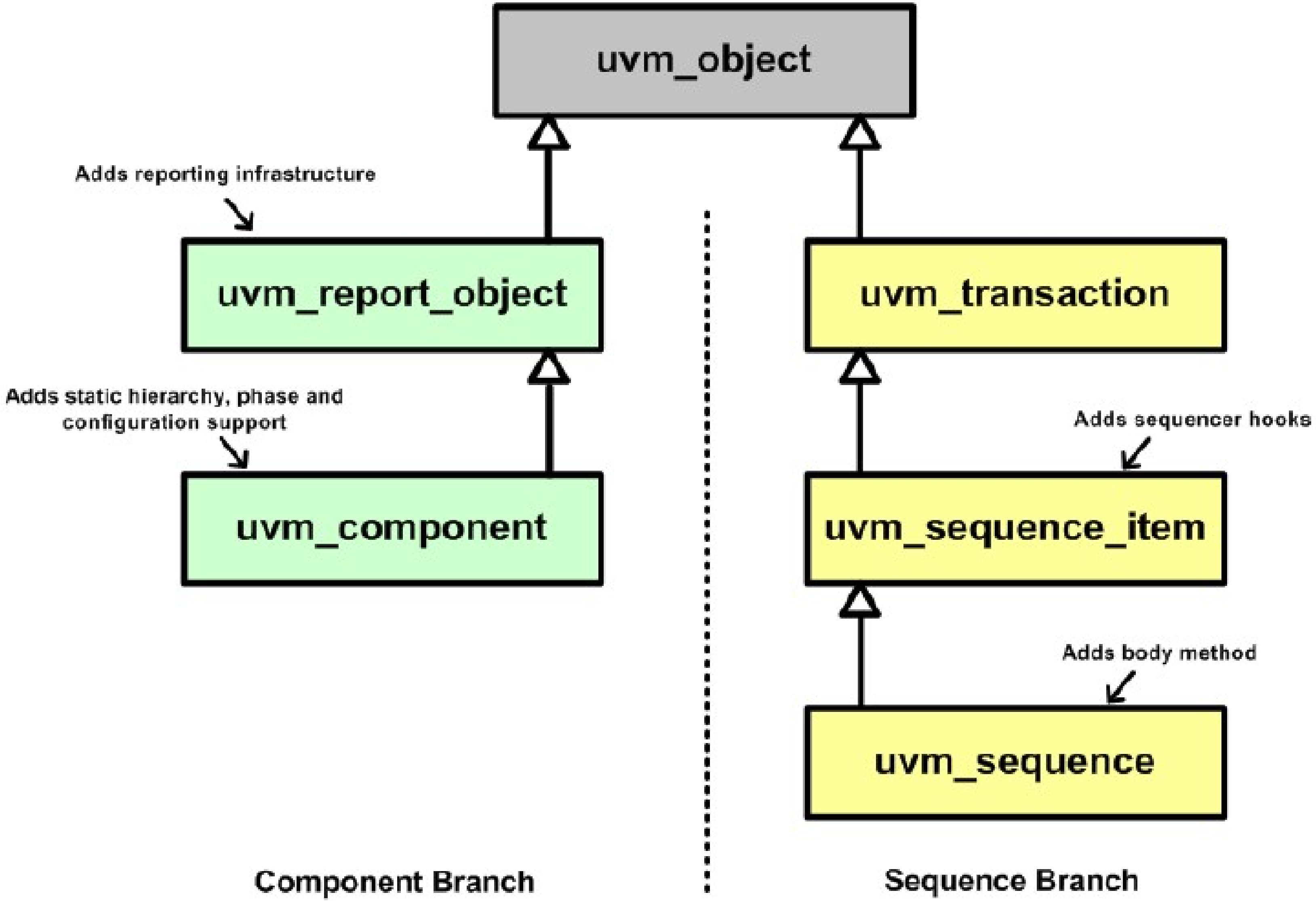
- □ The uvm_sequence_item – sometimes referred to as a transaction – is a uvm_object that contains the data fields necessary to implement the protocol and communicate with the DUT.
- □ The uvm_driver is responsible for converting the sequence_item(s) into “pin wiggles” on the signal-level interface to send and receive data to/from the DUT.
- □ The sequence_items are provided by one or more uvm_sequence objects that define stimulus at the transaction level and execute on the agent’s uvm_sequencer component.
- □ The sequencer is responsible for executing the sequences, arbitrating between them and routing sequence items between the driver and the sequence.

- The uvm_monitor is responsible for passively observing the pin-level behavior on the DUT interface, converting it into sequence items and providing those sequence items to analysis components in the agent or elsewhere in the testbench such as coverage collectors or scoreboards.
- UVM Agents also have a *configuration object* that allows the test writer to control aspects of the agent as the testbench is assembled and executed.
- UVM Agent isolates the testbench and the UVM Sequence from
- details of the interface implementation.\
- A sequence that provides data packets, for example, can be reused with different UVM Agents that may implement AHB, PCI or other protocols.
- A UVM testbench will typically have one agent per DUT interface.

- For a given design, the UVM Agents and other components are encapsulated in a uvm_env environment component, which is typically design-specific.
- Like an agent, an environment typically has a configuration object associated with it that allows the test to control aspects of the environment as well as to control the agents instantiated in the environment.
- Because environments are themselves UVM components, they can be assembled into a higher-level environment.
- As block-level designs are assembled into subsystems and systems, the block-level UVM environment associated with the block may be reused as a component in the subsystem-level environment, which can itself be reused in the system-level testbench.

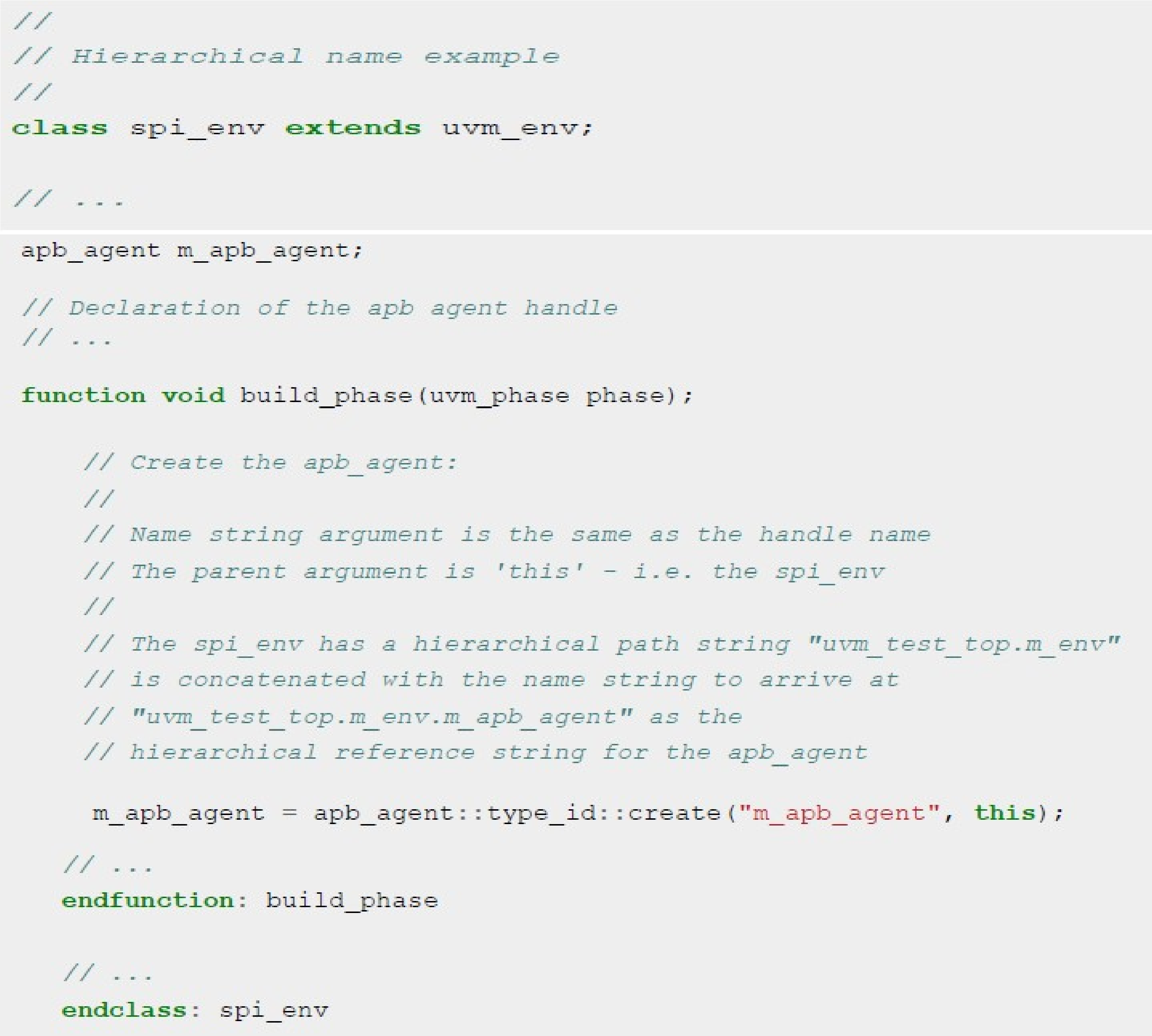
- □ Once the environment has been defined, the uvm_test will instantiate, configure and build the environment, including customizing key aspects of the overall testbench, including ..*choosing variations of components to be used in the environment ..*choosing UVM Sequences to be run either in the background or as the main portion of the test ..*defining configuration objects for the environment, sub-environment(s) (if any) and agent(s) in the testbench.
- □ The UVM test is started from an initial block in the top-level HVL module by calling run_test().

Simplified UVM Inheritance Diagram



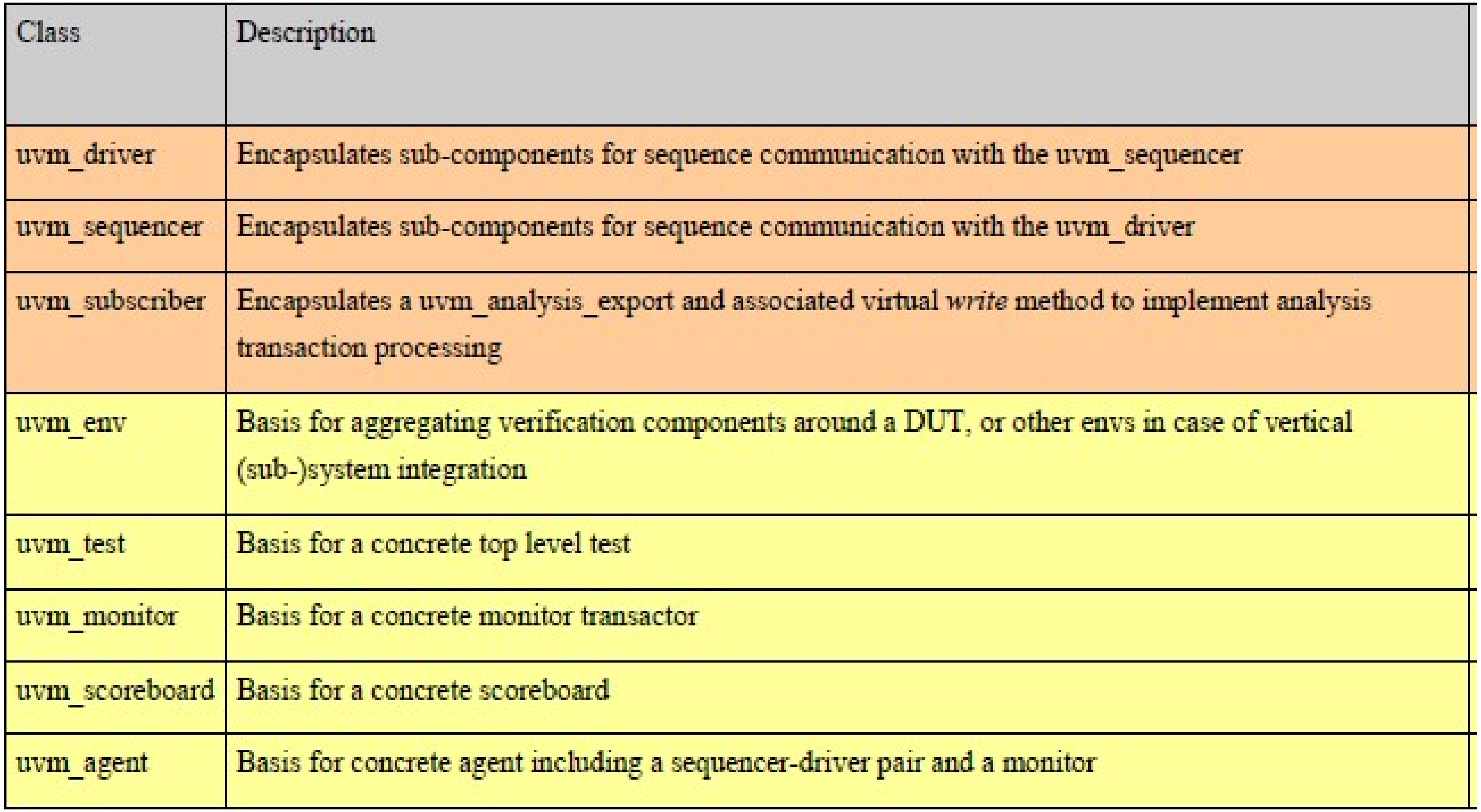
- A UVM testbench is composed of component objects extended from the uvm_component base class.
- When a uvm_component derived class object is created, it becomes part of the testbench hierarchy which persists for the duration of the simulation.
- This contrasts with the sequence branch of the UVM class hierarchy which involves transient objects - objects that are created, used and destroyed (i.e. garbage collected) once dereferenced

- in the code fragment below, an apb_agent component is created within the spi_env.
- Assuming the spi_env is instantiated in the top-level test component with the name "m_env," the hierarchical path name of the agent is the concatenation of the spi_env component's name, "uvm_test_top.m_env", the "dot" (".") operator, and the name passed as the first argument to the "create()" method, resulting in a hierarchical name for the agent of "uvm_test_top.m_env.m_apb_agent".
- Any references to the agent would need to use this string name.



```
//
// Hierarchical name example
//
class spi_env extends uvm_env;

// ...
 apb_agent m_apb_agent;

 // Declaration of the apb agent handle
 // ...

 function void build_phase(uvm_phase phase);

     // Create the apb_agent:
     //
     // Name string argument is the same as the handle name
     // The parent argument is 'this' - i.e. the spi_env
     //
     // The spi_env has a hierarchical path string "uvm_test_top.m_env"
     // is concatenated with the name string to arrive at
     // "uvm_test_top.m_env.m_apb_agent" as the
     // hierarchical reference string for the apb_agent

      m_apb_agent = apb_agent::type_id::create("m_apb_agent", this);

    // ...
    endfunction: build_phase

    // ...
    endclass: spi_env
```

- □ In order to provide flexibility in configuration and to allow the UVM testbench hierarchy to be built in an intelligent way, uvm_components are registered with the UVM factory.
- □ When a UVM component is created during the build phase, the factory is used to construct the component object.
- □ The UVM factory enables a component to be swapped for another of a compatible, derived type using a factory override.
- □ This is a useful technique for altering the functionality of a testbench without changing the testbench source code directly, which would require recompilation
- □ and hinder reuse.
- □ There are a number of coding conventions required for the factory to work and these are outlined in the article on the UVM Factory.

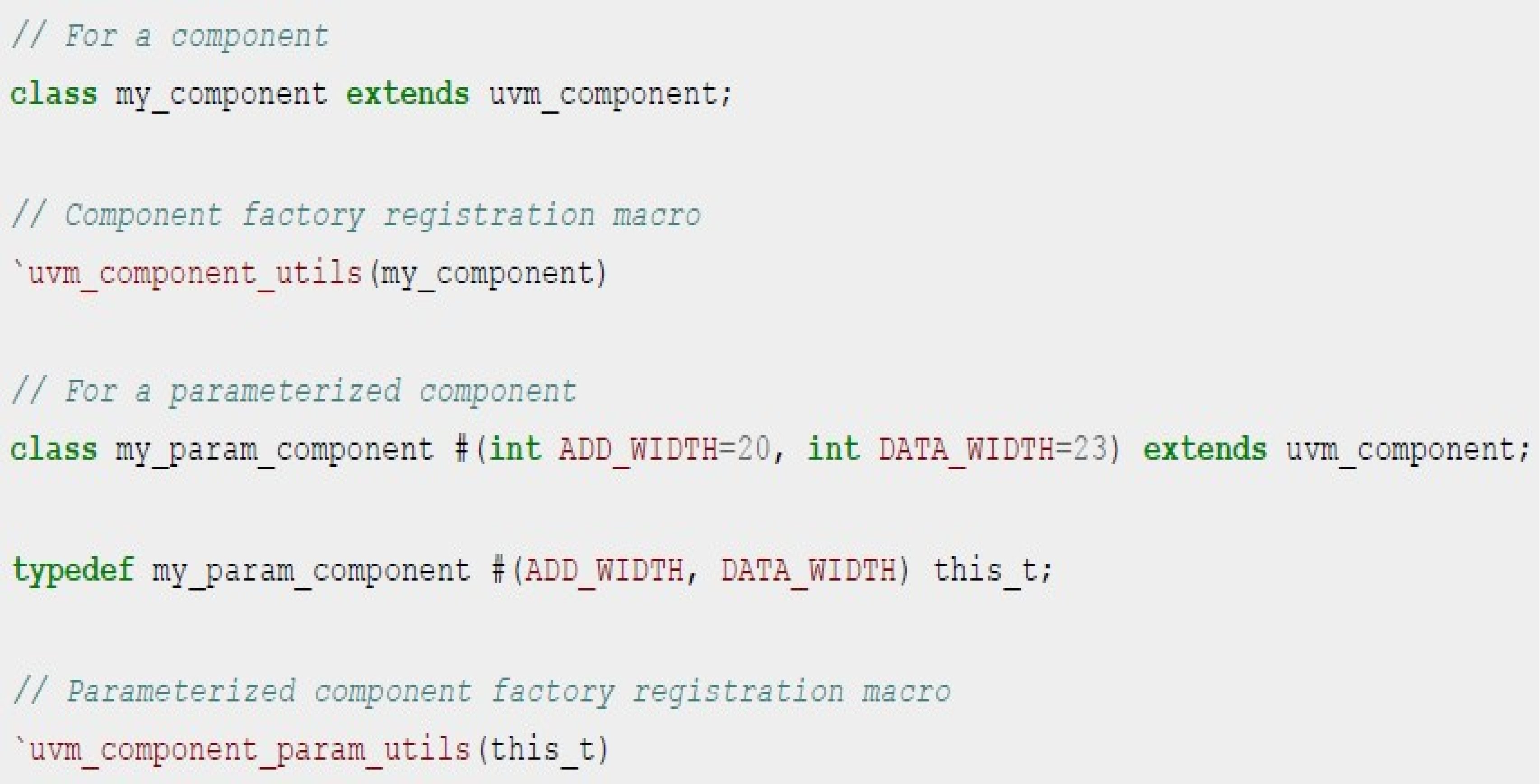


Class	Description
uvm_driver	Encapsulates sub-components for sequence communication with the uvm_sequencer
uvm_sequencer	Encapsulates sub-components for sequence communication with the uvm_driver
uvm_subscriber	Encapsulates a uvm_analysis_export and associated virtual *write* method to implement analysis transaction processing
uvm_env	Basis for aggregating verification components around a DUT, or other envs in case of vertical (sub-)system integration
uvm_test	Basis for a concrete top level test
uvm_monitor	Basis for a concrete monitor transactor
uvm_scoreboard	Basis for a concrete scoreboard
uvm_agent	Basis for concrete agent including a sequencer-driver pair and a monitor

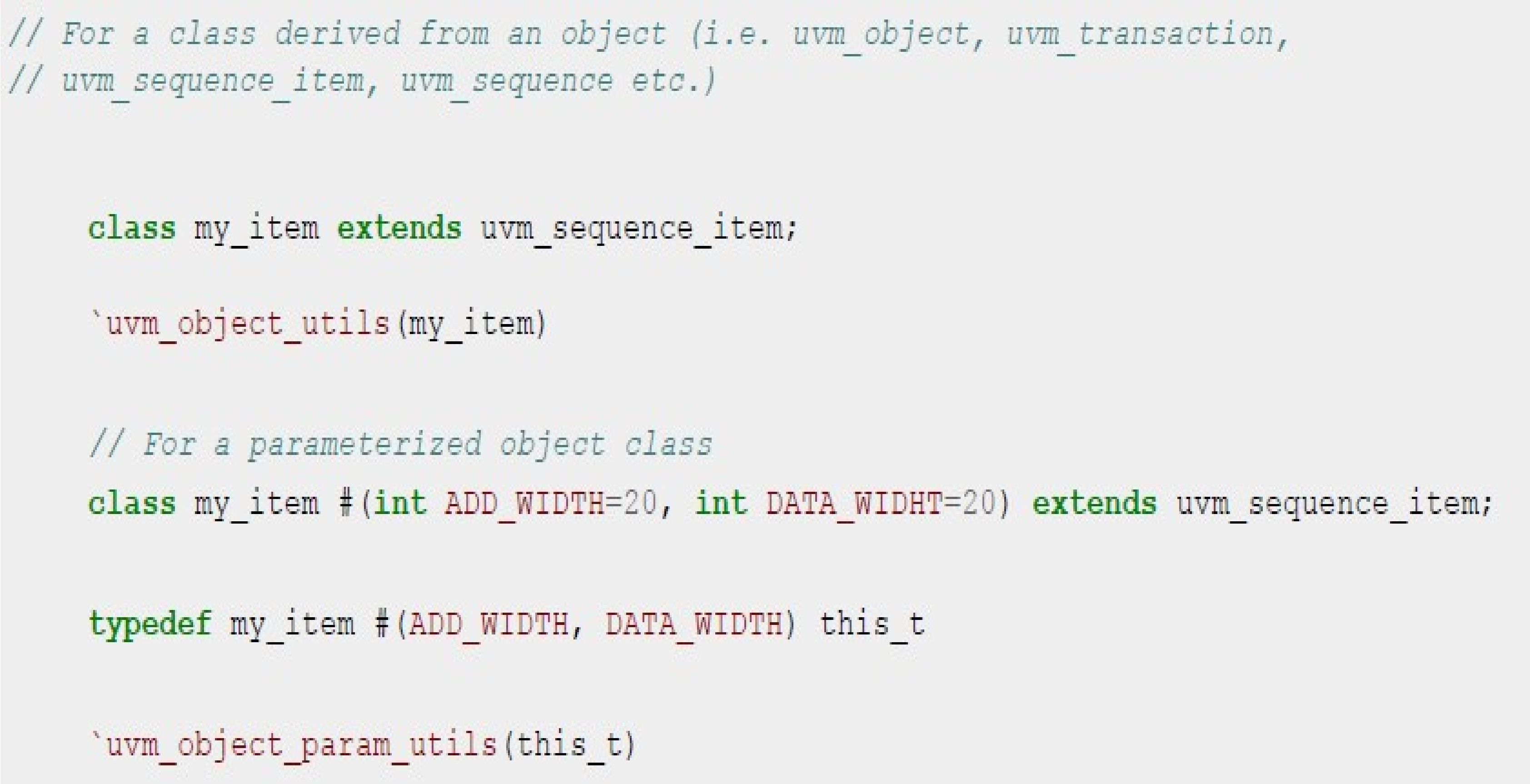
UVM factory

- □ The purpose of the UVM factory is to enable an object of one type to be substituted with an object of a derived type without changing the testbench structure or even the testbench code.
- □ The mechanism used is referred to as an override, by either instance or type.
- □ This functionality is very handy for changing sequence behavior or replacing one version of a component by another.
- □ Any two components to be swapped must be polymorphically compatible.
- □ This includes the requirement that all the same TLM interface handles and TLM objects must be created by the replacement component.

Registration

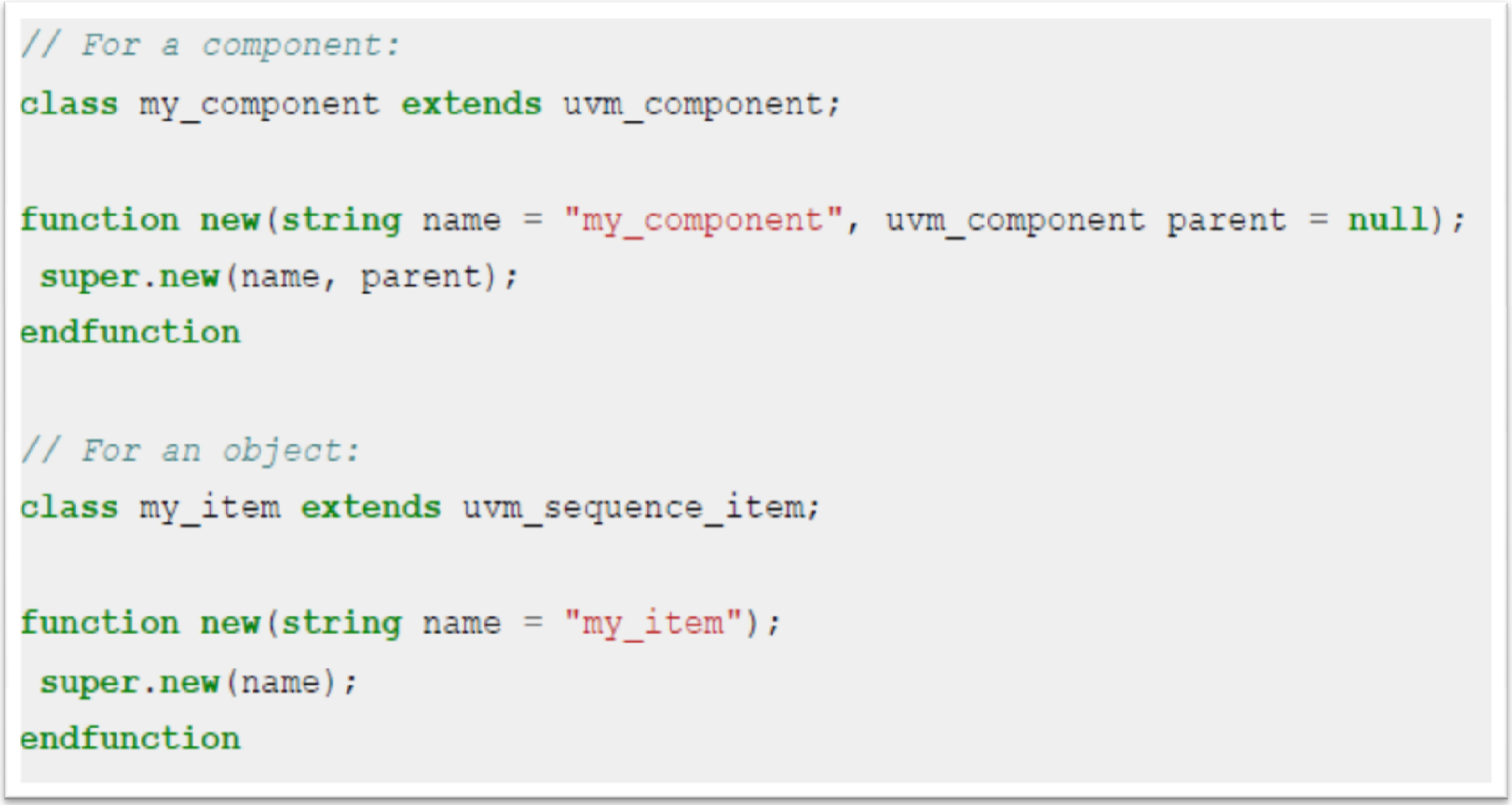


```
// For a component
class my_component extends uvm_component;

// Component factory registration macro
`uvm_component_utils(my_component)

// For a parameterized component
class my_param_component #(int ADD_WIDTH=20, int DATA_WIDTH=23) extends uvm_component;

typedef my_param_component #(ADD_WIDTH, DATA_WIDTH) this_t;

// Parameterized component factory registration macro
`uvm_component_param_utils(this_t)
```



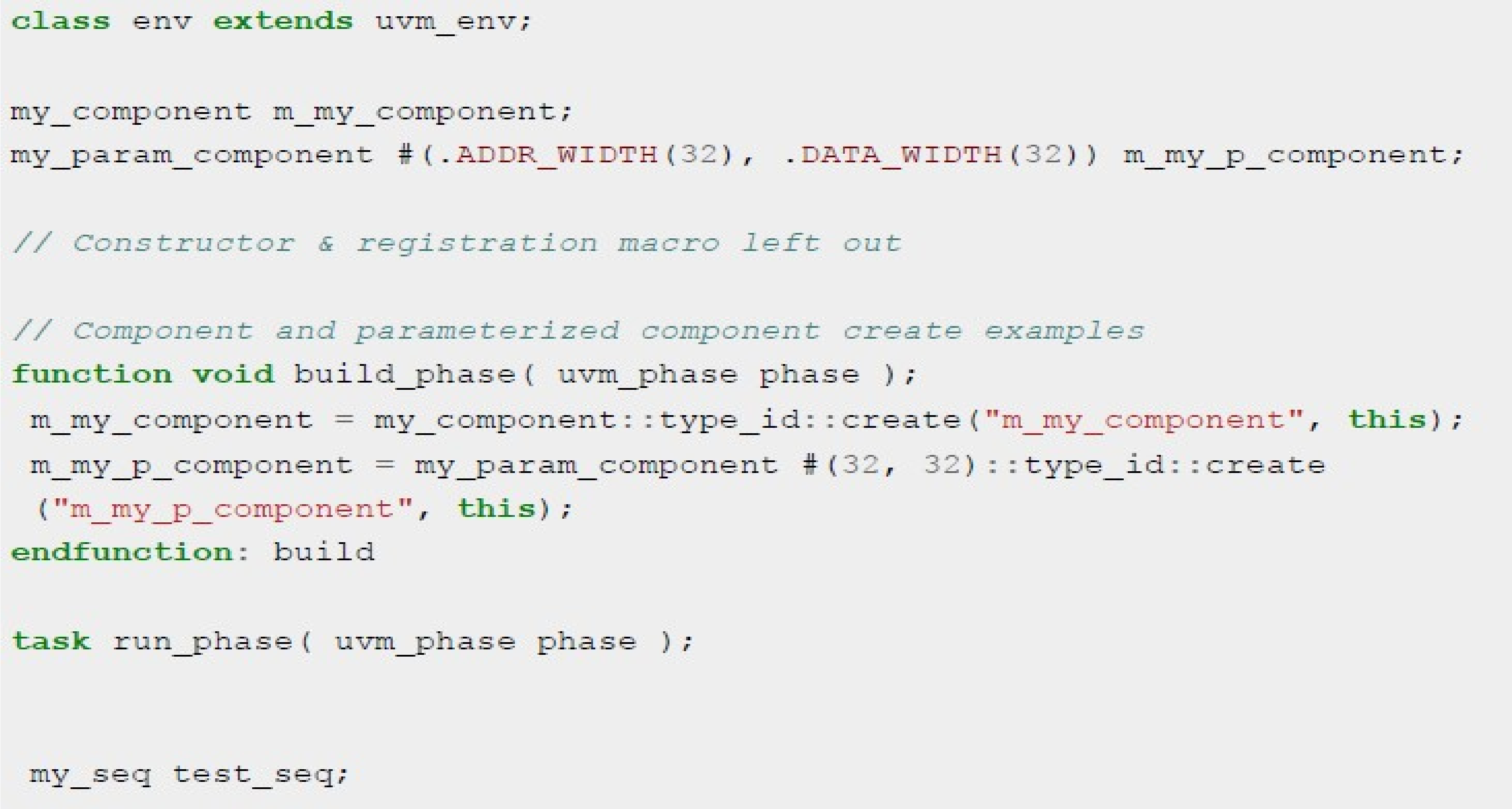
```
// For a class derived from an object (i.e. uvm_object, uvm_transaction,
// uvm_sequence_item, uvm_sequence etc.)


    class my_item extends uvm_sequence_item;

    `uvm_object_utils(my_item)

    // For a parameterized object class
    class my_item #(int ADD_WIDTH=20, int DATA_WIDHT=20) extends uvm_sequence_item;

    typedef my_item #(ADD_WIDTH, DATA_WIDTH) this_t

    `uvm_object_param_utils(this_t)
```

Constructor Defaults



```
// For a component:
class my_component extends uvm_component;

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


// For an object:
class my_item extends uvm_sequence_item;

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

Component and Object Creation



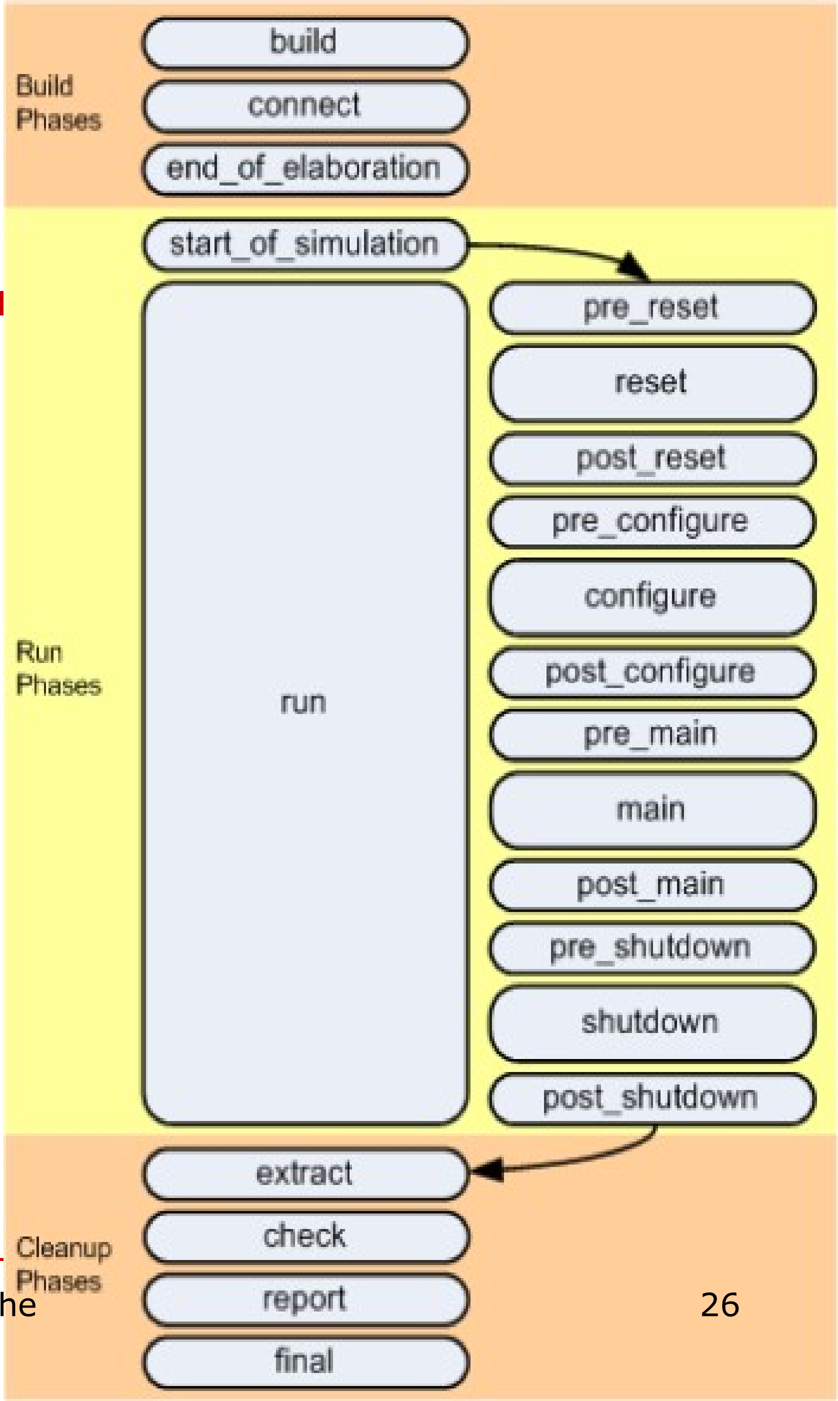
```
class env extends uvm_env;

my_component m_my_component;
my_param_component #(.ADDR_WIDTH(32), .DATA_WIDTH(32)) m_my_p_component;

// Constructor & registration macro left out

// Component and parameterized component create examples
function void build_phase( uvm_phase phase );
 m_my_component = my_component::type_id::create("m_my_component", this);
 m_my_p_component = my_param_component #(32, 32)::type_id::create
  ("m_my_p_component", this);
endfunction: build

task run_phase( uvm_phase phase );


 my_seq test_seq;
```

Standard UVM Phases

1. Build phases - where the testbench is configured and constructed

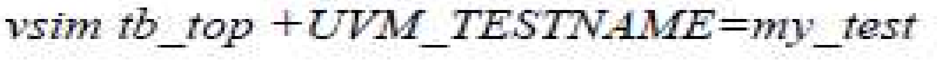
2. Run-time phases - where time is consumed in running the testcase on the testbench

3. Clean up phases - where the results of the testcase are collected and reported



Starting UVM Phase Execution

- □ To start a UVM testbench, the run_test() method has to be called from the static part of the testbench. It is usually called from within an initial block in the top level module of the testbench.
- □ Calling run_test() constructs the UVM environment root component and then initiates the UVM phasing.
- □ The run_test() method can be passed a string argument to define the default type name of an uvm_component derived class which is used as the root node of the testbench hierarchy.
- □ However, the run_test() method checks for a command line plusarg called UVM_TESTNAME and uses that plusarg string to lookup a factory registered uvm_component, overriding any default type name.



```
vsim tb_top +UVM_TESTNAME=my_test
```

- □ **Build Phases**
- □ The build phases are executed at the start of the UVM testbench simulation and their overall purpose is to construct, configure and connect the testbench component hierarchy.
- □ All the build phase methods are functions and therefore execute in zero simulation time.

- **build**
- Once the UVM testbench root node component is constructed, the build phase starts to execute.
- It constructs the testbench component hierarchy from the top downwards.
- The construction of each component is deferred so that each layer in the component hierarchy can be configured by the level above.
- During the build phase uvm_components are indirectly constructed using the UVM factory.

- **connect**
- The connect phase is used to make TLM connections between components or to assign handles to testbench resources.
- It has to occur after the build method has put the testbench component hierarchy in place and works from the bottom of the hierarchy upwards.

- **end_of_elaboration**
- The end_of_elaboration phase is used to make any final adjustments to the structure, configuration or connectivity of the testbench before simulation starts.
- Its implementation can assume that the testbench component hierarchy and inter-connectivity is in place.
- This phase executes bottom up.

- **Run Time Phases**
- The testbench stimulus is generated and executed during the run time phases which follow the build phases.
- After the start_of_simulation phase, the UVM executes the run phase and the phases pre_reset through to post_shutdown in parallel.
- Run phase is a phase that transactors will use.
- The other phases were added to the UVM to give finer runtime phase granularity for tests, scoreboards and other similar components.
- It is expected that most testbenches will only use reset, configure, main and shutdown and not their pre and post variants

- **start_of_simulation**
- The start_of_simulation phase is a function which occurs before the time consuming part of the testbench begins.
- It is intended to be used for displaying banners; testbench topology; or configuration information.
- It is called in bottom up order.
- **run**
- The run phase occurs after the start_of_simulation phase and is used for the stimulus generation and checking activities of the testbench.
- The run phase is implemented as a task, and all uvm_component run_phase() tasks are executed in parallel.
- Transactors such as drivers and monitors will nearly always use this phase.

- **Parallel Run-Time Phases**
- NOTE: The following run-time phases execute in-order, in parallel with the run_phase phase.
- These phases should only be called from the test and the env to start sequences.
- Drivers, monitors and other components should not implement these phases.

- **pre_reset**
- The pre_reset phase starts at the same time as the run phase. Its purpose is to take care of any activity that should occur before reset, such as waiting for a power-good signal to go active.
- **reset**
- The reset phase is reserved for DUT or interface specific reset behavior. For example, this phase would be used to generate a reset and to put an interface into its default state.
- **post_reset**
- The post_reset phase is intended for any activity required immediately following reset. This might include training or rate negotiation behaviour.

- **pre_configure**
- The pre_configure phase is intended for anything that is required to prepare for the DUT's configuration process after reset is completed, such as waiting for components (e.g. drivers) required for configuration to complete training
- and/or rate negotiation. It may also be used as a last chance to modify the information described by the test/environment to be uploaded to the DUT.
- **configure**
- The configure phase is used to program the DUT and any memories in the testbench so that it is ready for the start of the test case. It can also be used to set signals to a state ready for the test case start.
- **post_configure**
- The post_configure phase is used to wait for the effects of configuration to propagate through the DUT, or for it to reach a state where it is ready to start the main test stimulus.

- ☐ **main**
- ☐ This is where the stimulus specified by the test case is generated and applied to the DUT. It completes when either all stimulus is exhausted or a timeout occurs. Most data throughput will be handled by sequences started in this phase.
- ☐ **shutdown**
- ☐ The shutdown phase is used to ensure that the effects of the stimulus generated during the main phase have propagated through the DUT and that any resultant data has drained away.
- ☐ It might also be used to execute time consuming sequences that read status registers.

Clean Up Phases

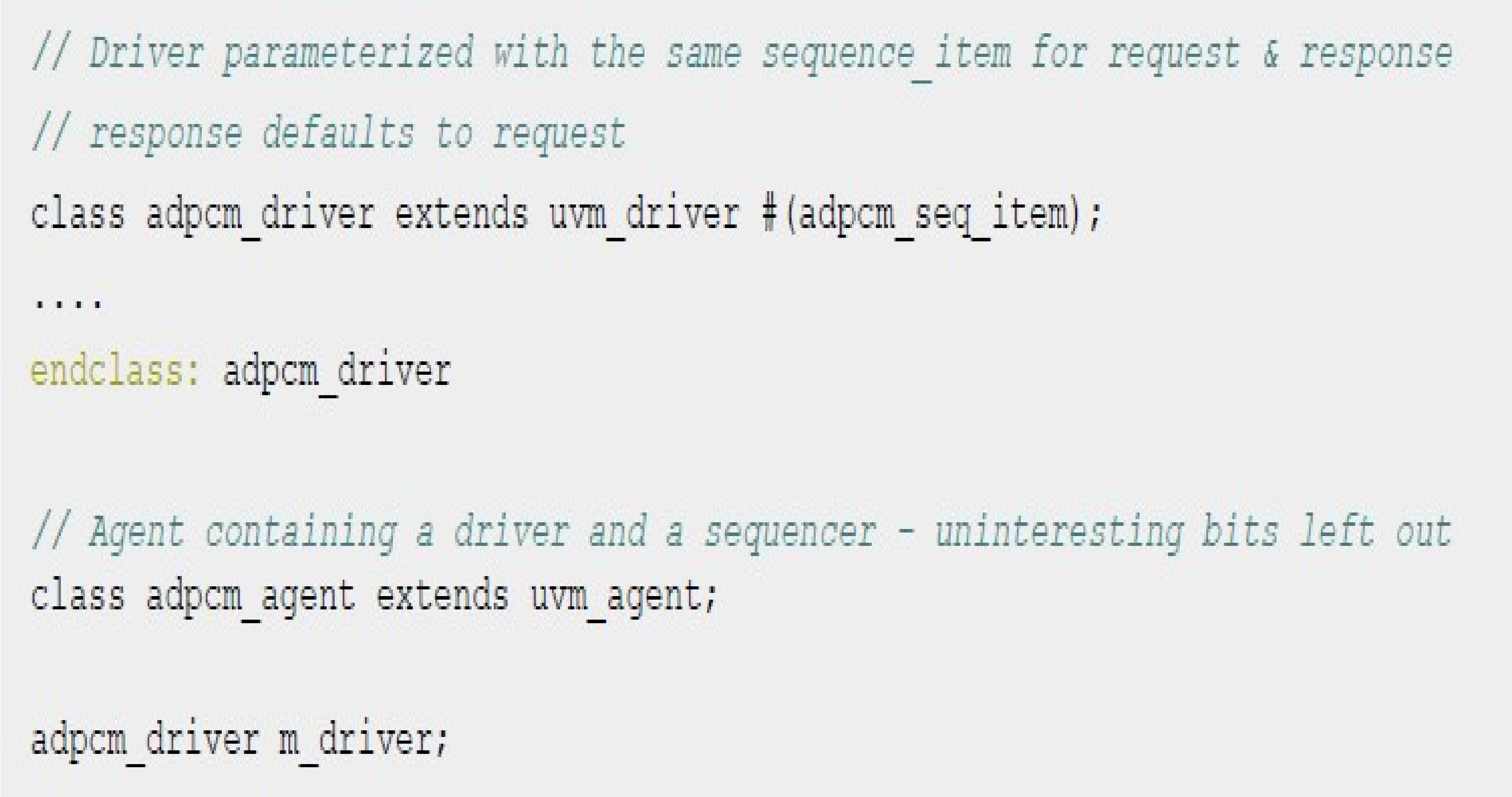
- **extract**
- The extract phase is used to retrieve and process information from scoreboards and functional coverage monitors.
- This may include the calculation of statistical information used by the report phase. This phase is usually used by analysis components.
- **check**
- The check phase is used to check that the DUT behaved correctly and to identify any errors that may have occurred during the execution of the testbench. This phase is usually used by analysis components.
- **report**
- The report phase is used to display the results of the simulation or to write the results to file. This phase is usually used by analysis components.
- **final**
- The final phase is used to complete any other outstanding actions that the testbench has not already completed.

DRIVER

- □ The UVM driver is responsible for communicating at the transaction level with the sequence via TLM communication with the sequencer and converting between the sequence_item on the transaction side and pin-level activity in communicating with the DUT via a virtual interface.
- □ As the name implies, drivers typically get a sequence_item and use that information to drive signals to the DUT and may, in certain applications, also receive a pin-level response from the DUT and convert it back into a sequence_item for the sequence to complete the transaction.
- □ A driver may also function as a "responder" (i.e. in "slave mode") in which the driver reacts to pin-level activity in the interface to communicate with a sequence that then sends a response transaction back to the driver to complete the protocol transaction

- □ A user-defined driver component is a proxy class derived from a *uvm_driver* base class and contains a BFM which is a SystemVerilog interface.
- □ The *uvm_driver* base class provides a seq_item_port that gets connected by the agent to the seq_item_export of the agent's *uvm_sequencer*.
- □ Usually, responses are passed back to the sequence through the seq_item_port as well, but certain applications may require that responses be sent back to the sequencer via the *rsp_port* of the driver

- □ The *uvm_driver* is designed ultimately to interact with a *uvm_sequence* running on the connected *uvm_sequencer*



```
// Driver parameterized with the same sequence_item for request & response
// response defaults to request
class adpcm_driver extends uvm_driver #(adpcm_seq_item);
....
endclass: adpcm_driver


// Agent containing a driver and a sequencer - uninteresting bits left out
class adpcm_agent extends uvm_agent;

adpcm_driver m_driver;
```

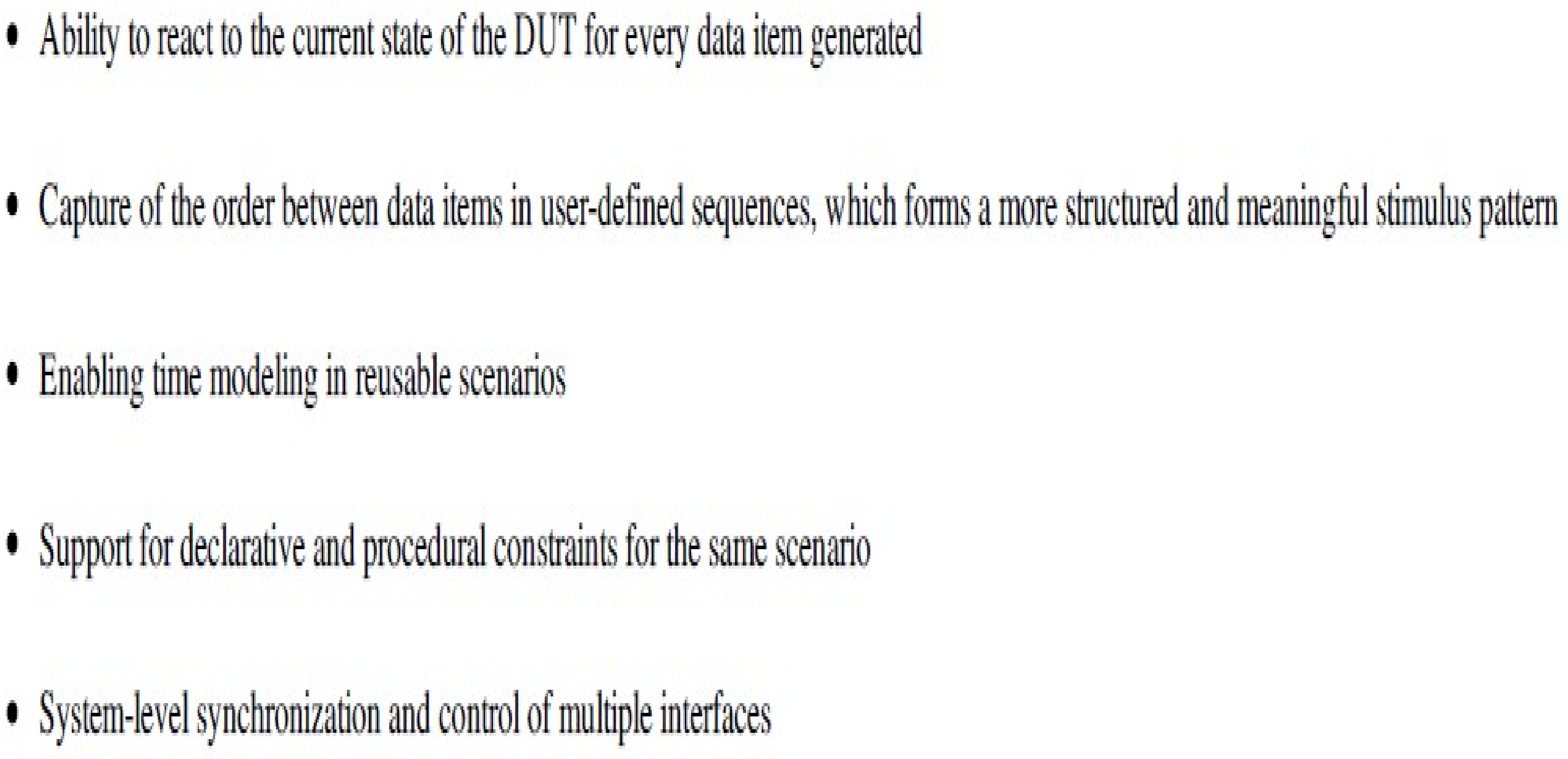
```
class protocol_driver extends uvm_driver #(protocol_seq_item);

    ‘uvm_component_utils(protocol_driver);

    virtual protocol_interface vif;
    // Virtual interface declaration
    ....
    function new(string name, uvm_componenet parent);
        super.new(name, parent);
    endfunction

    extern task run_phase(uvm_phase phase);

endclass
```

Driver :: Run_phase

```
task protocol_driver::run_phase(uvm_phase phase);
   forever begin
            #10 seq_item_port.get(req);
            …
            vif.addr = req.addr;
            vif.data = req.data;
   end
endtask
```

Sequencer

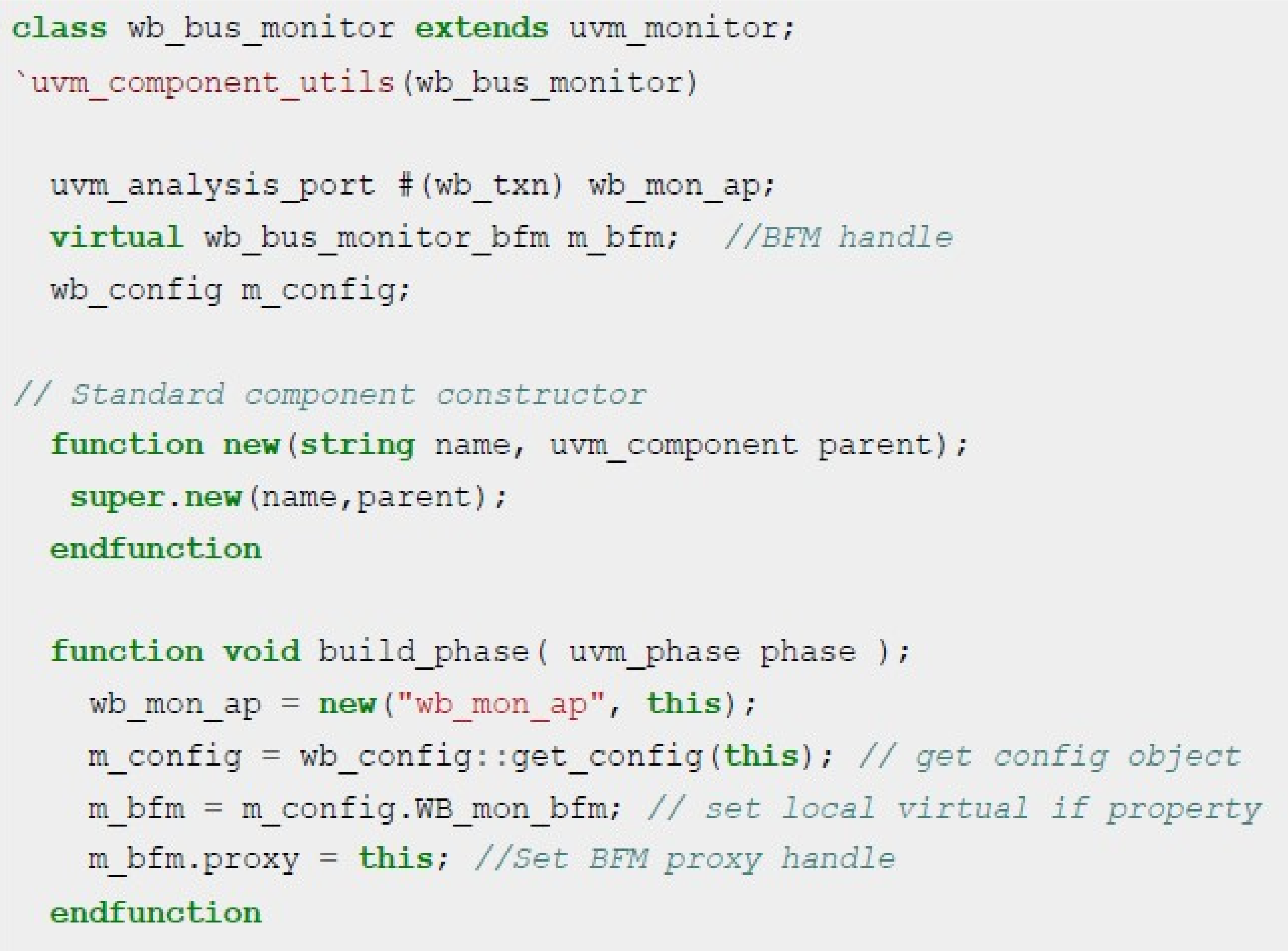
- A sequencer is an advanced stimulus generator that controls the items provided to the driver for execution.
- By default, a sequencer behaves similarly to a simple stimulus generator and returns a random data item upon request from the driver.
- This default behavior allows you to add constraints to the data item class in order to control the distribution of randomized values.
- Unlike generators that randomize arrays of transactions or one transaction at a time, a sequencer includes many important built-in features.



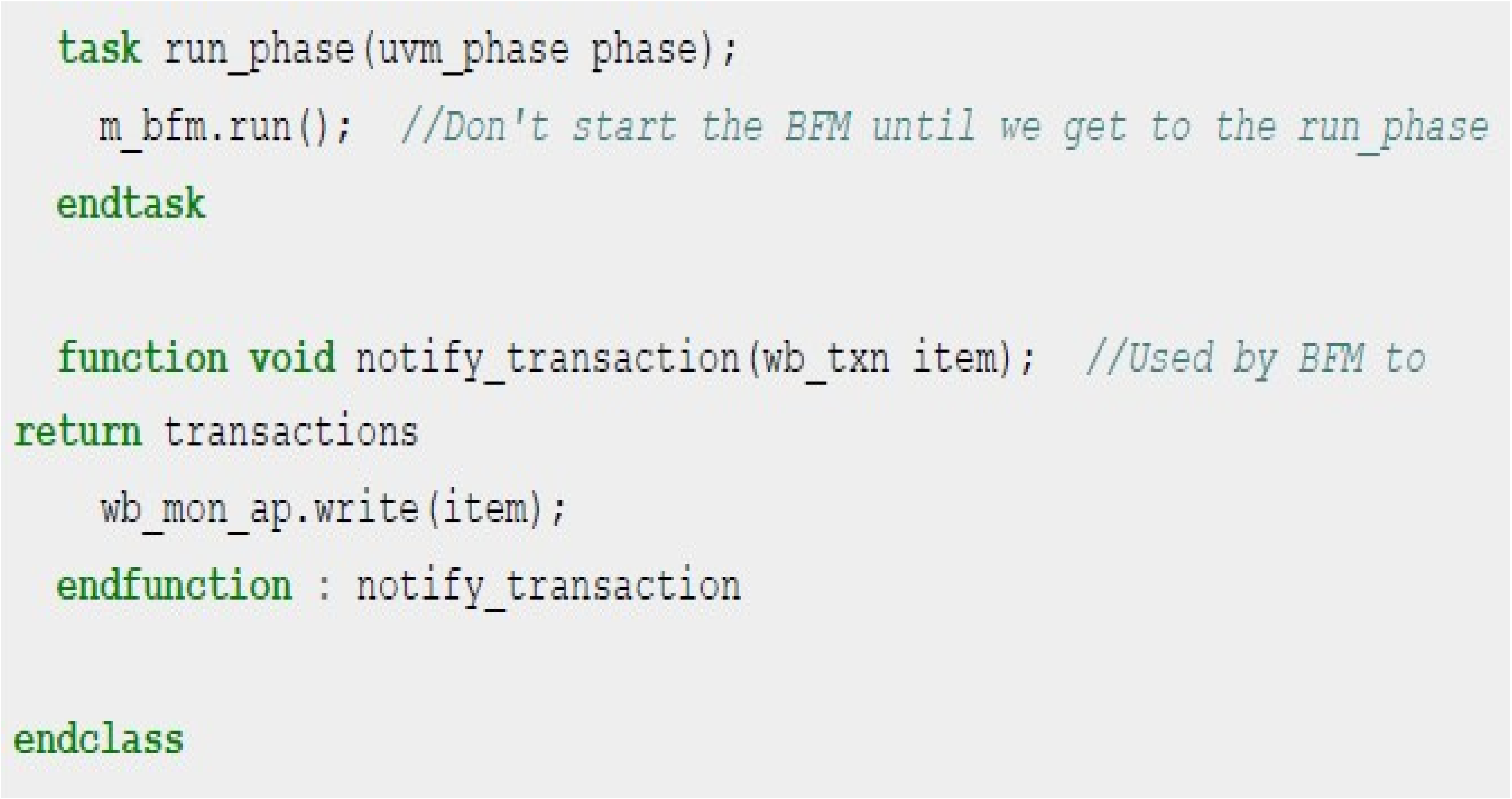
- Ability to react to the current state of the DUT for every data item generated
- Capture of the order between data items in user-defined sequences, which forms a more structured and meaningful stimulus pattern
- Enabling time modeling in reusable scenarios
- Support for declarative and procedural constraints for the same scenario
- System-level synchronization and control of multiple interfaces

UVM Monitor

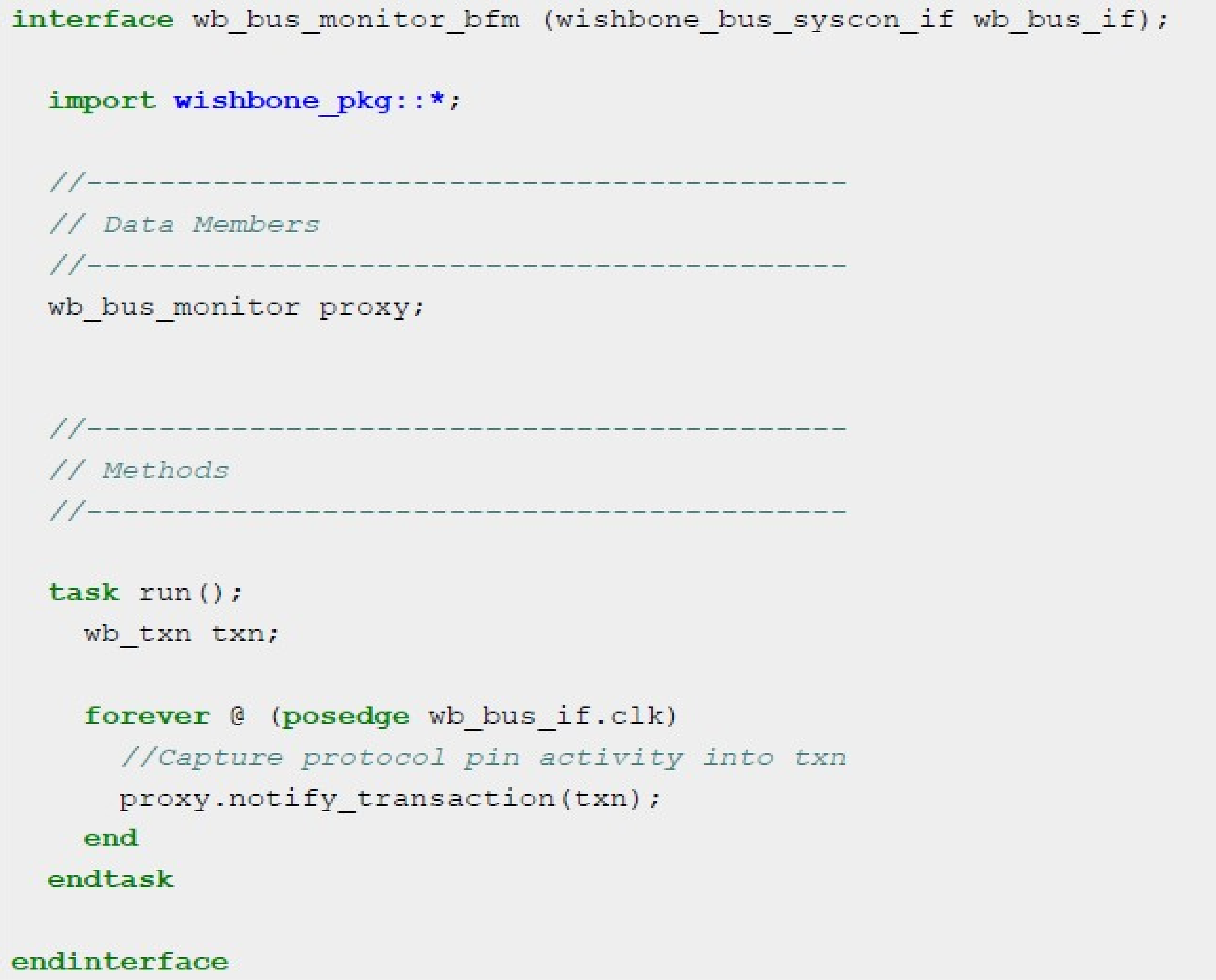
- The key difference between a Monitor and a Driver is that a Monitor is always passive. It does not drive any signals on the interface.
- When an agent is placed in passive mode, the Monitor continues to execute.
- It contains code that recognizes protocol patterns in the signal activity.
- Once a protocol pattern is recognized, a Monitor builds an abstract transaction model representing that activity, and broadcasts the transaction to any interested components.
- Monitors are composed of a proxy class which should extend from uvm_monitor and a BFM which is a SystemVerilog interface.
- The proxy should have one analysis port and a virtual interface handle that points to a BFM interface.



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

  uvm_analysis_port #(wb_txn) wb_mon_ap;
  virtual wb_bus_monitor_bfm m_bfm;  //BFM handle
  wb_config m_config;

// Standard component constructor
  function new(string name, uvm_component parent);
   super.new(name,parent);
  endfunction

  function void build_phase( uvm_phase phase );
    wb_mon_ap = new("wb_mon_ap", this);
    m_config = wb_config::get_config(this); // get config object
    m_bfm = m_config.WB_mon_bfm; // set local virtual if property
    m_bfm.proxy = this; //Set BFM proxy handle
  endfunction
```



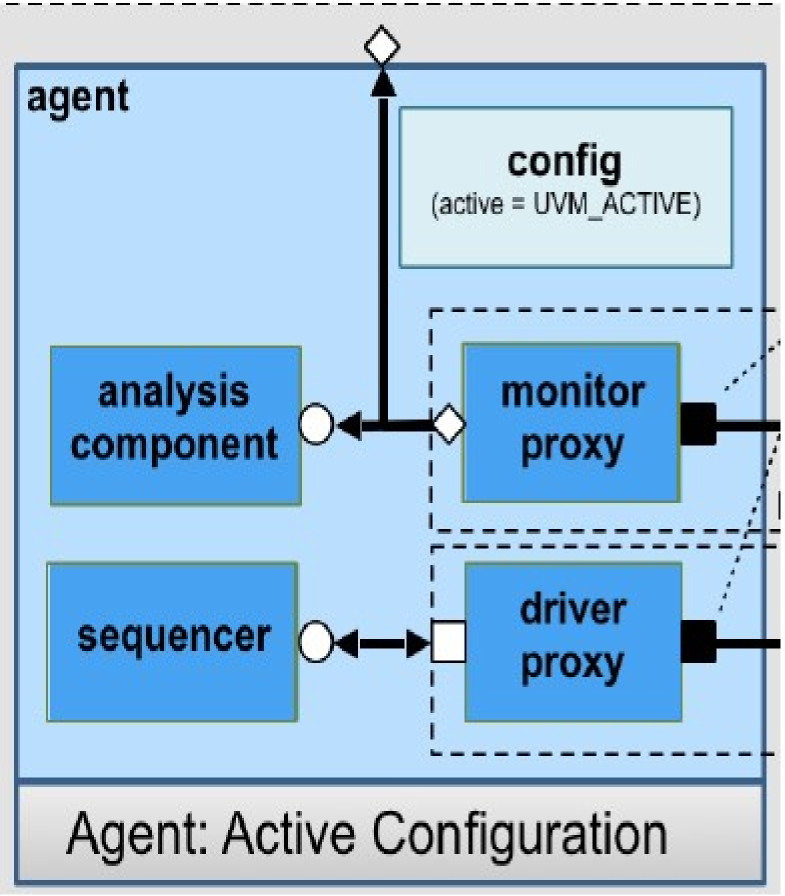
```
  task run_phase(uvm_phase phase);
    m_bfm.run();  //Don't start the BFM until we get to the run_phase
  endtask

  function void notify_transaction(wb_txn item);  //Used by BFM to
return transactions
    wb_mon_ap.write(item);
  endfunction : notify_transaction

endclass
```

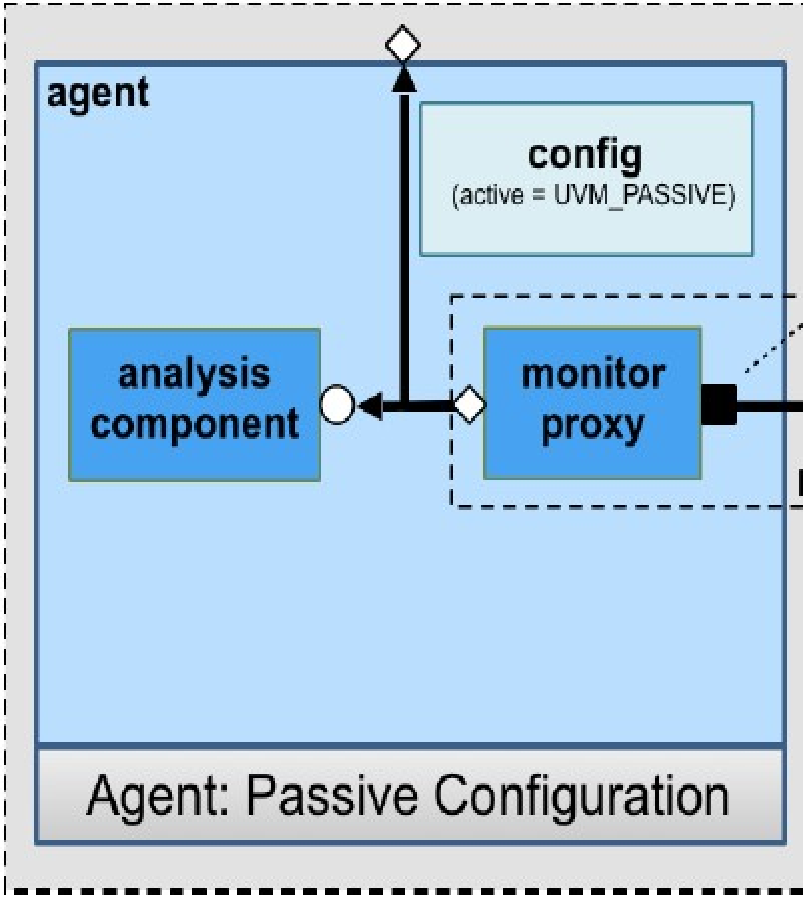


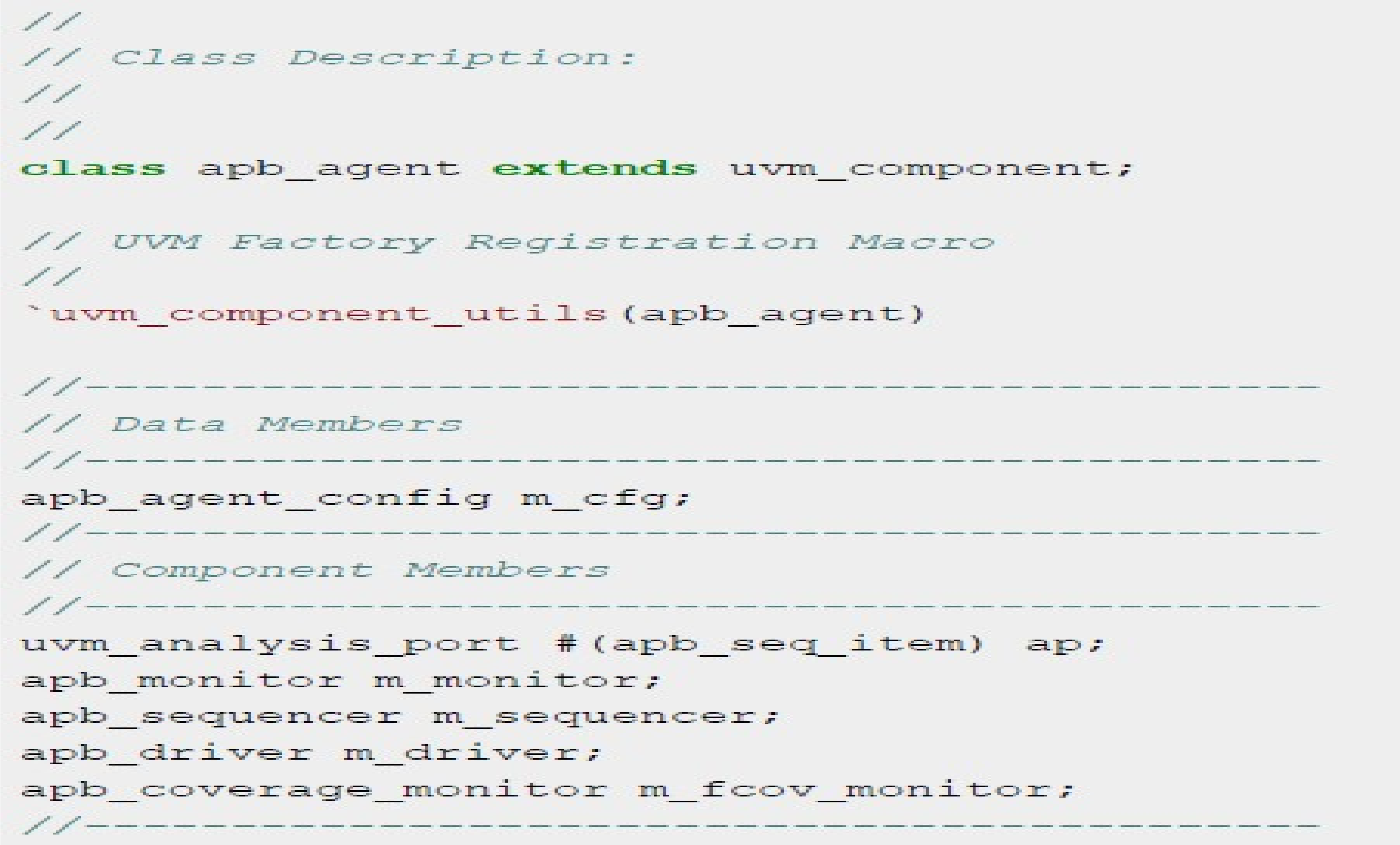
```
interface wb_bus_monitor_bfm (wishbone_bus_syscon_if wb_bus_if);

  import wishbone_pkg::*;

  //------------------------------------------
  // Data Members
  //------------------------------------------
  wb_bus_monitor proxy;


  //------------------------------------------
  // Methods
  //------------------------------------------

  task run();
    wb_txn txn;

    forever @ (posedge wb_bus_if.clk)
      //Capture protocol pin activity into txn
      proxy.notify_transaction(txn);
    end
  endtask

endinterface
```

UVM Agent

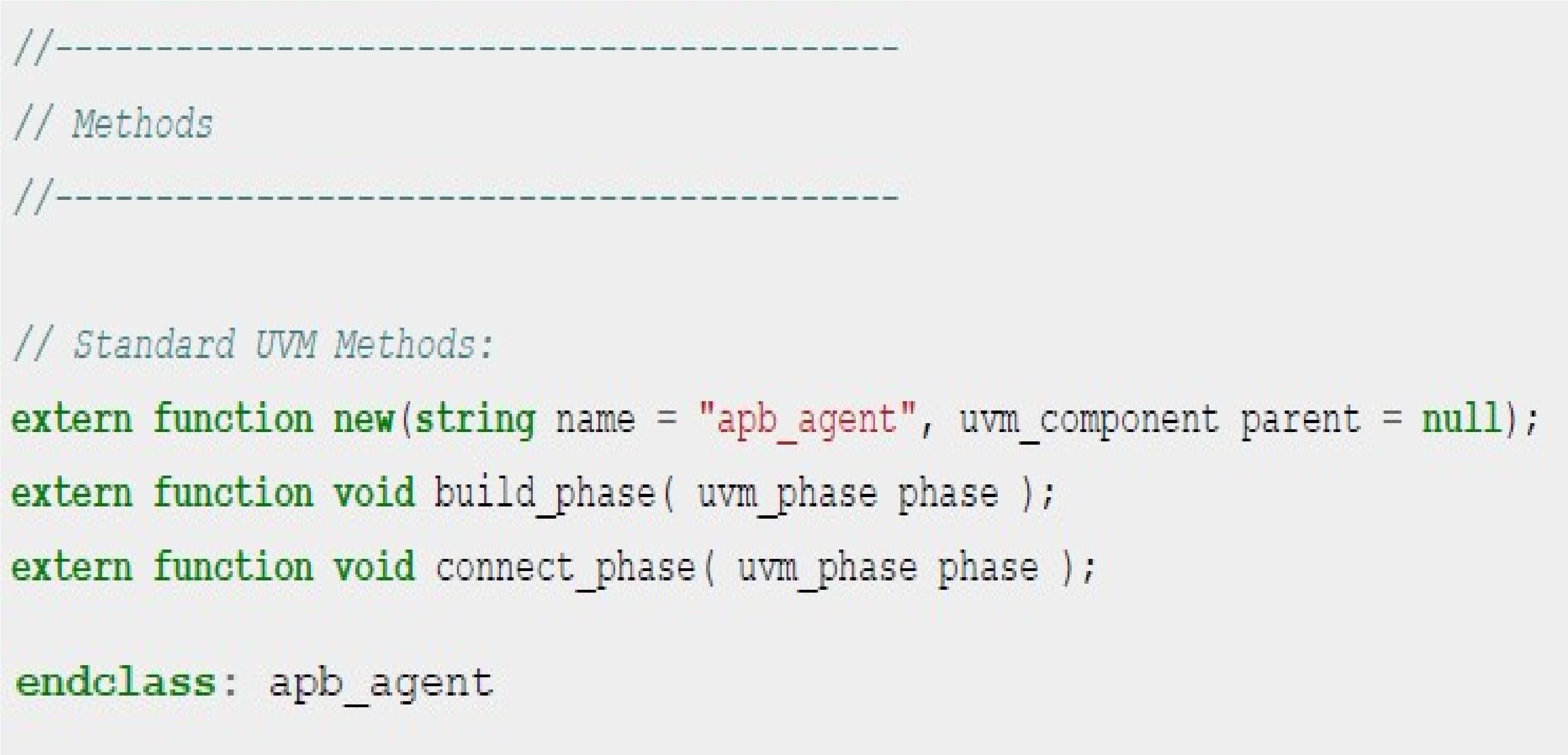
- □ Sequencers, drivers, monitors, and collectors can be reused independently, but this requires the environment integrator to learn the names, roles, configuration, and hookup of each of these entities.
- □ Agents encapsulate a driver, sequencer, monitor.
- □ UVCs can contain more than one agent.
- □ Some agents are proactive (for example, master or transmit agents) and initiate transactions to the DUT, while other agents (slave or receive agents) react to transaction requests. Agents should be configurable so that they can be either active or passive.
- □ Active agents emulate devices and drive transactions according to test directives. Passive agents only monitor DUT activity.


agent
config
(active = UVM_ACTIVE)
analysis component
monitor proxy
sequencer
driver proxy
Agent: Active Configuration

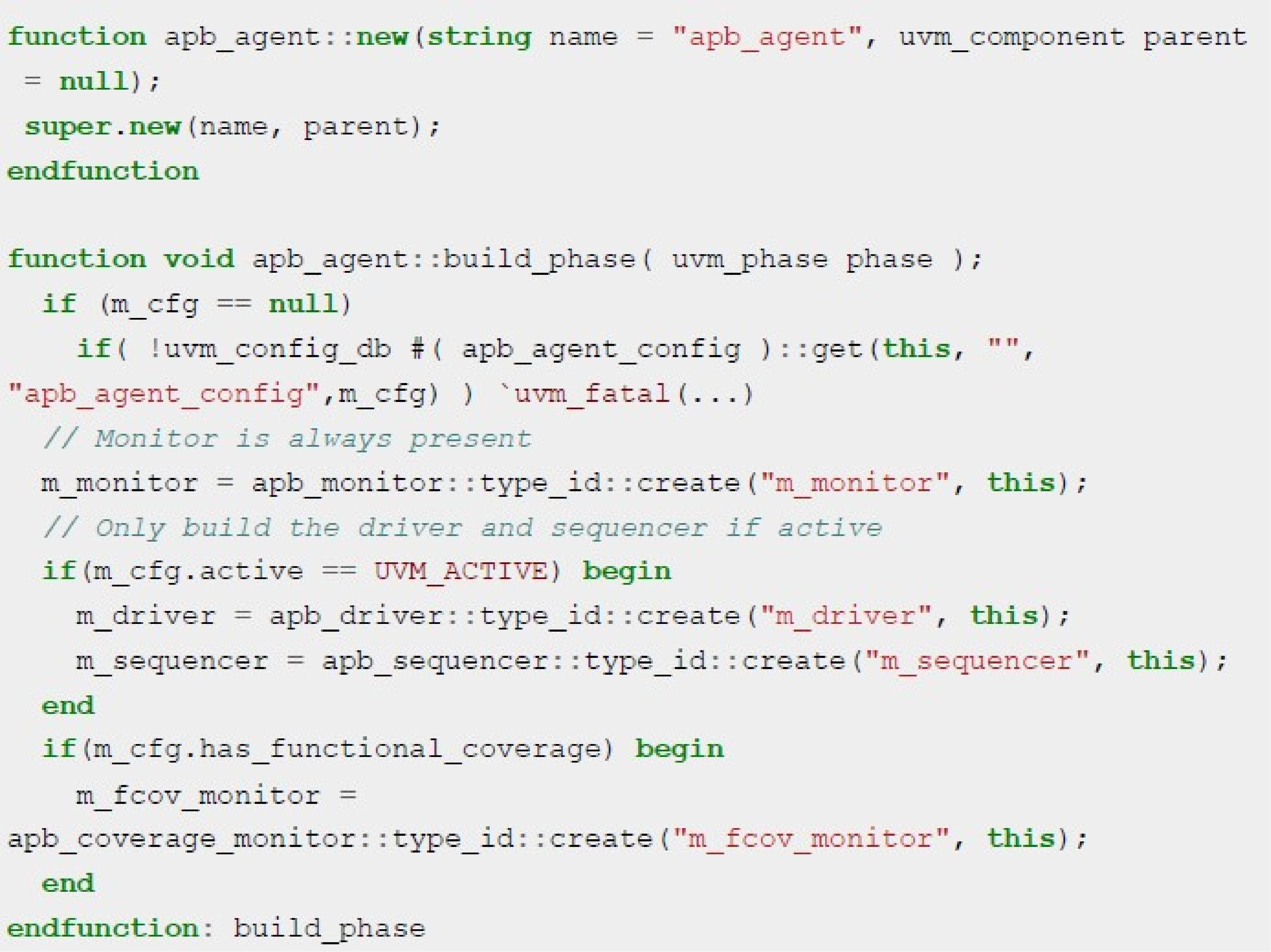

agent
config
(active = UVM_PASSIVE)
analysis component
monitor proxy
Agent: Passive Configuration



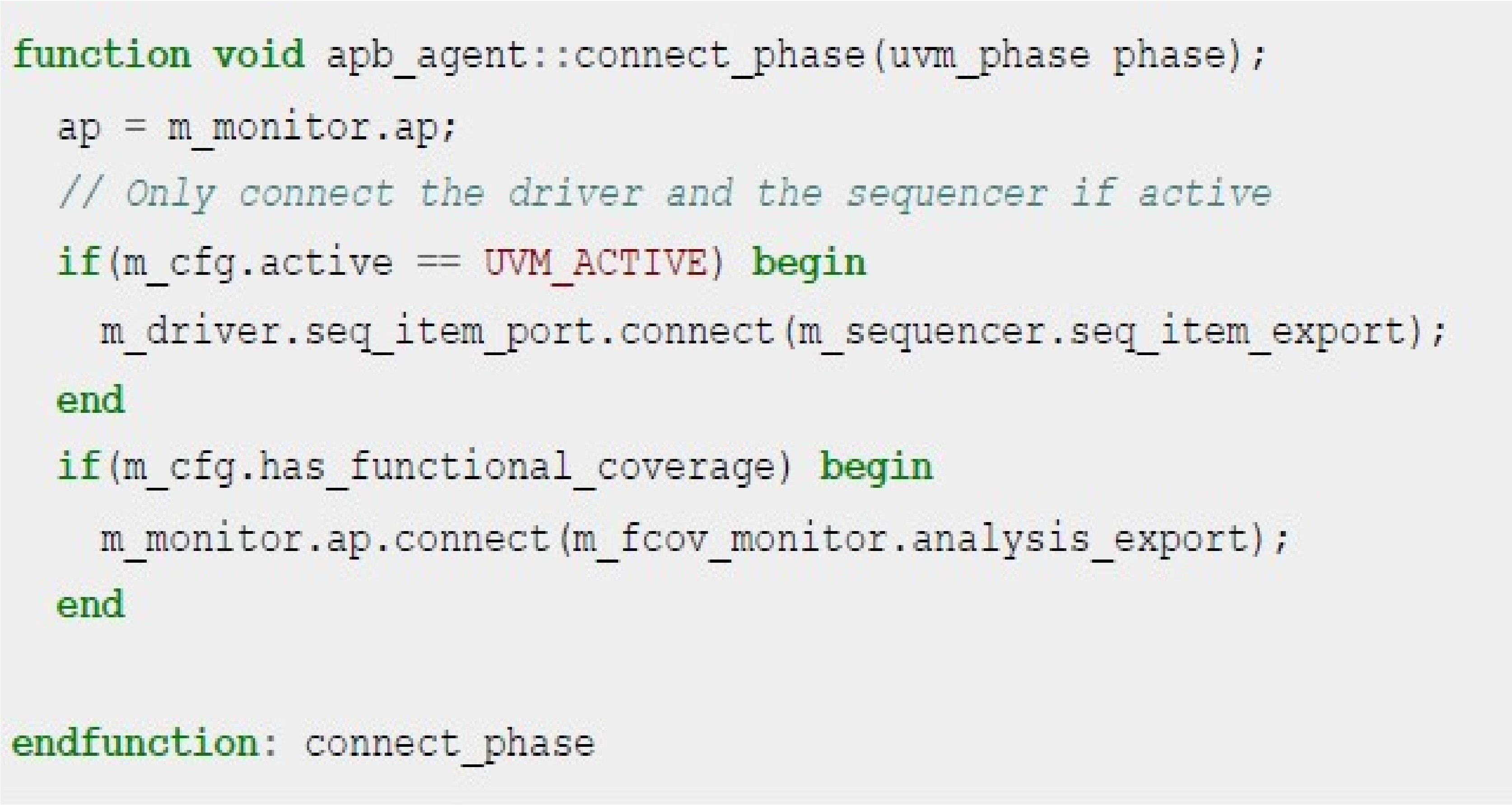
```
//
// Class Description:
//
//
class apb_agent extends uvm_component;

// UVM Factory Registration Macro
//
`uvm_component_utils(apb_agent)

//-------------------------------------------
// Data Members
//-------------------------------------------
apb_agent_config m_cfg;
//-------------------------------------------
// Component Members
//-------------------------------------------
uvm_analysis_port #(apb_seq_item) ap;
apb_monitor m_monitor;
apb_sequencer m_sequencer;
apb_driver m_driver;
apb_coverage_monitor m_fcov_monitor;
//-------------------------------------------
```



```
//------------------------------------------
// Methods
//------------------------------------------

// Standard UVM Methods:
extern function new(string name = "apb_agent", uvm_component parent = null);
extern function void build_phase( uvm_phase phase );
extern function void connect_phase( uvm_phase phase );

endclass: apb_agent
```



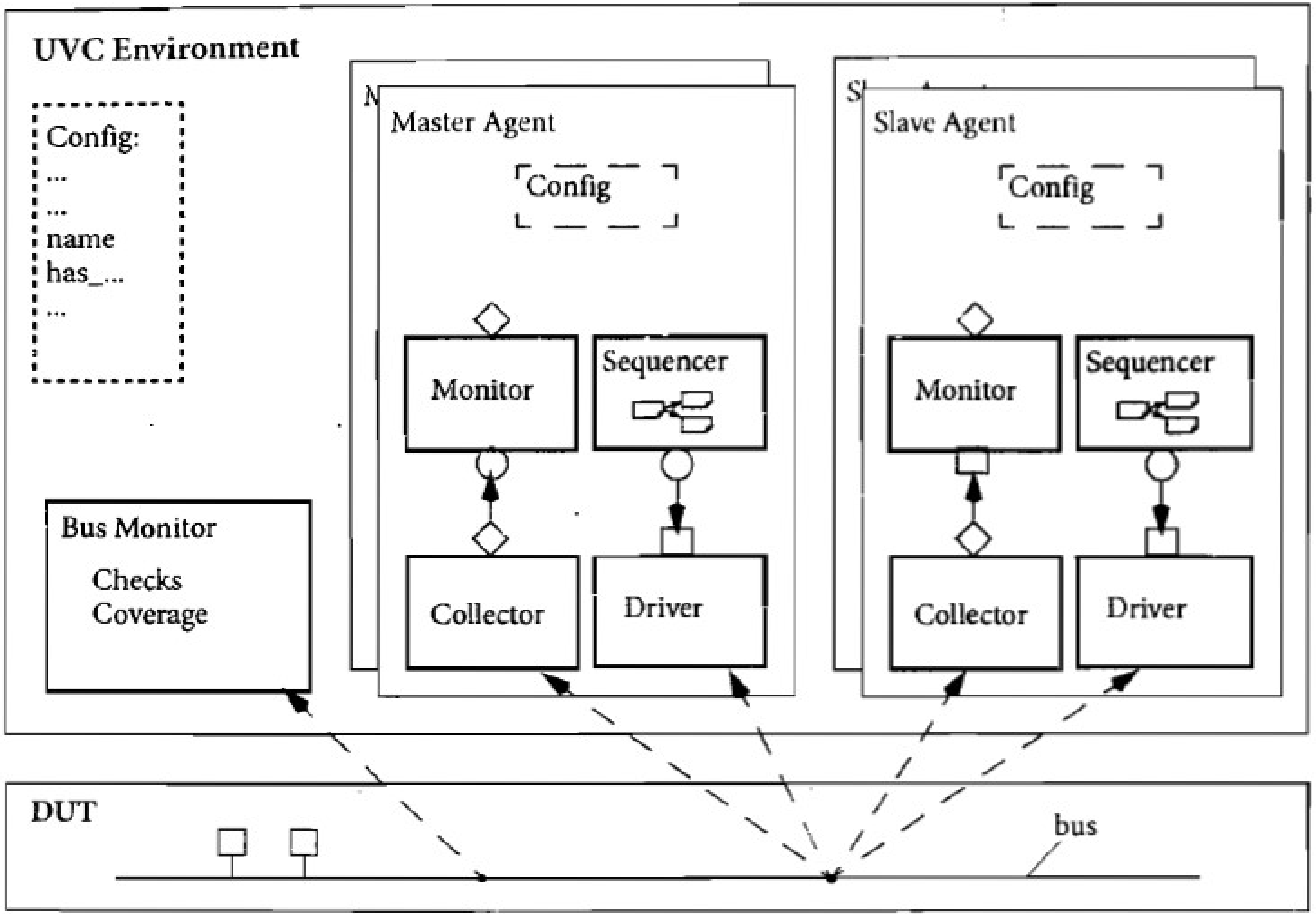
```
function apb_agent::new(string name = "apb_agent", uvm_component parent
 = null);
 super.new(name, parent);
endfunction

function void apb_agent::build_phase( uvm_phase phase );
  if (m_cfg == null)
    if( !uvm_config_db #( apb_agent_config )::get(this, "",
"apb_agent_config",m_cfg) ) `uvm_fatal(...)
  // Monitor is always present
  m_monitor = apb_monitor::type_id::create("m_monitor", this);
  // Only build the driver and sequencer if active
  if(m_cfg.active == UVM_ACTIVE) begin
    m_driver = apb_driver::type_id::create("m_driver", this);
    m_sequencer = apb_sequencer::type_id::create("m_sequencer", this);
  end
  if(m_cfg.has_functional_coverage) begin
    m_fcov_monitor =
apb_coverage_monitor::type_id::create("m_fcov_monitor", this);
  end
endfunction: build_phase
```



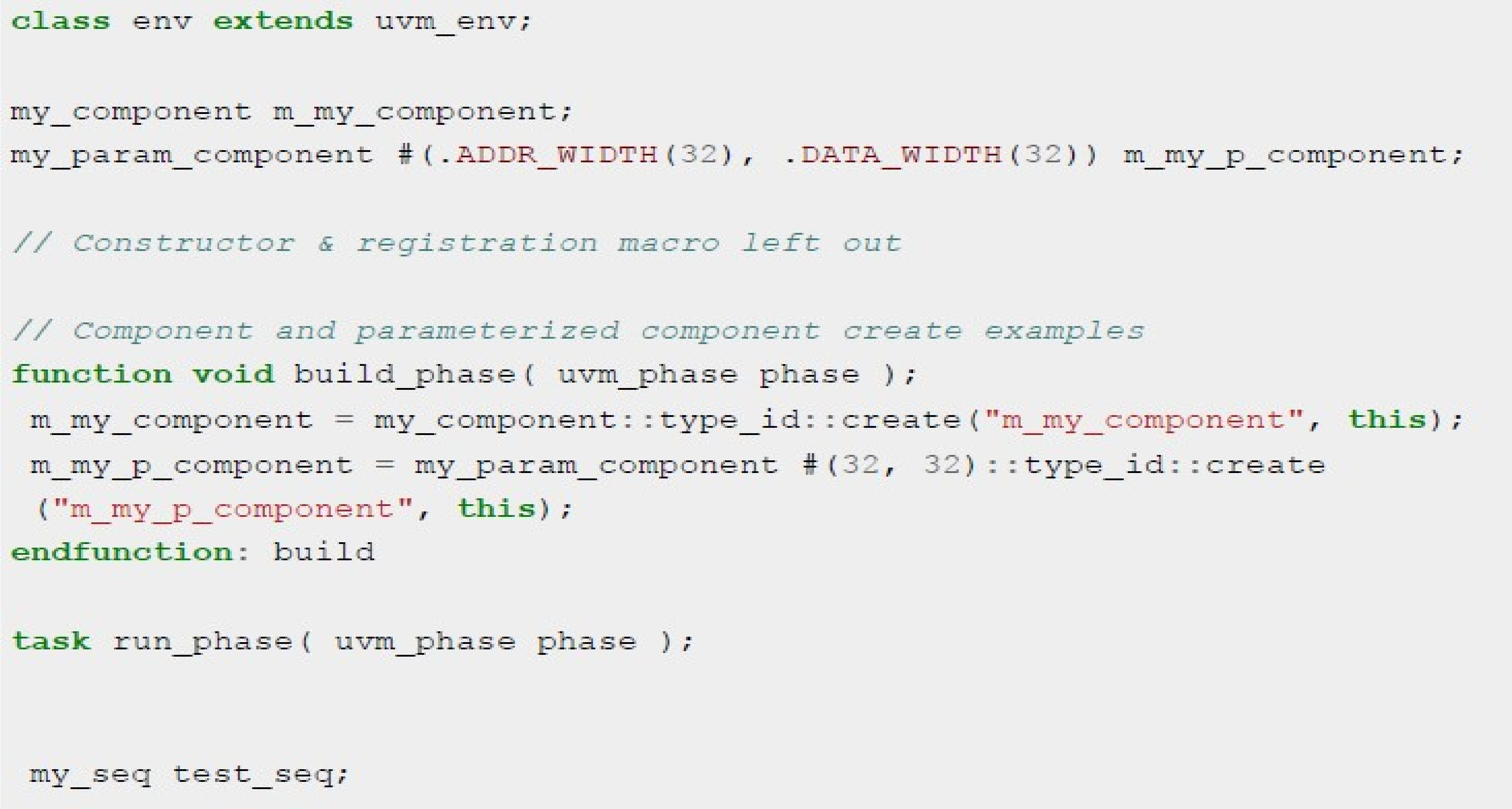
```
function void apb_agent::connect_phase(uvm_phase phase);
  ap = m_monitor.ap;
  // Only connect the driver and the sequencer if active
  if(m_cfg.active == UVM_ACTIVE) begin
    m_driver.seq_item_port.connect(m_sequencer.seq_item_export);
  end
  if(m_cfg.has_functional_coverage) begin
    m_monitor.ap.connect(m_fcov_monitor.analysis_export);
  end

endfunction: connect_phase
```

The Environment

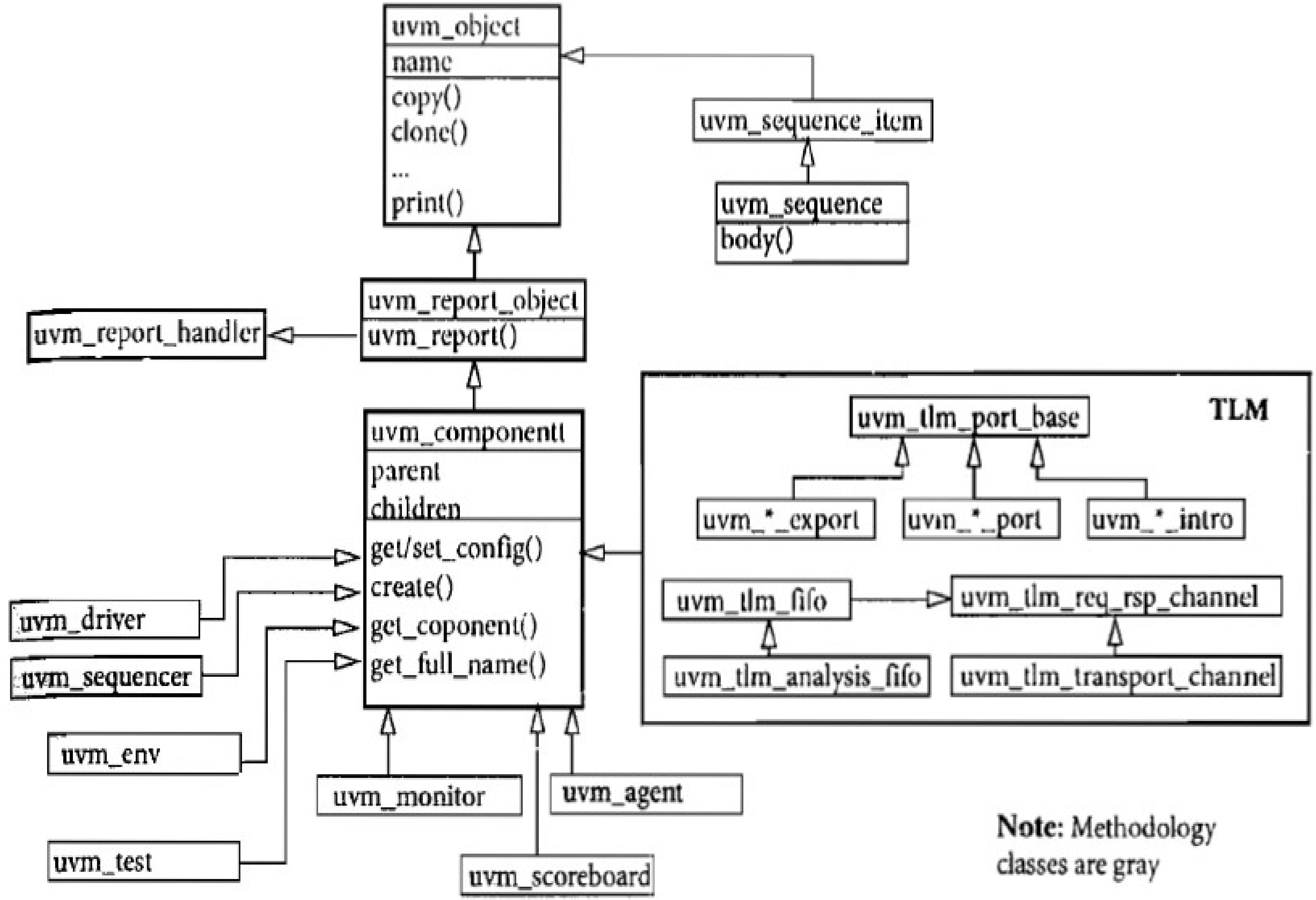
- □ The environment (env) is the top-level component of the UVC.
- □ It contains one or more agents, as well as other components such as a bus monitor.
- □ The env contains configuration properties that enable you to customize the topology and behavior to make it reusable.
- □ For example, active agents can be changed into passive agents when the verification environment is reused for system verification.
- □ The environment class (uvm_env) is designed to provide a flexible, reusable, and extendable verification component.
- □ The main function of the environment class is to model behavior by generating constrained-random traffic, monitoring DUT responses, checking the validity of the protocol activity, and collecting coverage.
- □ You can use derivation to specialize the existing classes to their specific protocol.


UVC Environment
Config:
...
...
name
has_...
...
Bus Monitor
Checks
Coverage
Master Agent
Config
Monitor
Sequencer
Collector
Driver
Slave Agent
Config
Monitor
Sequencer
Collector
Driver
DUT
bus

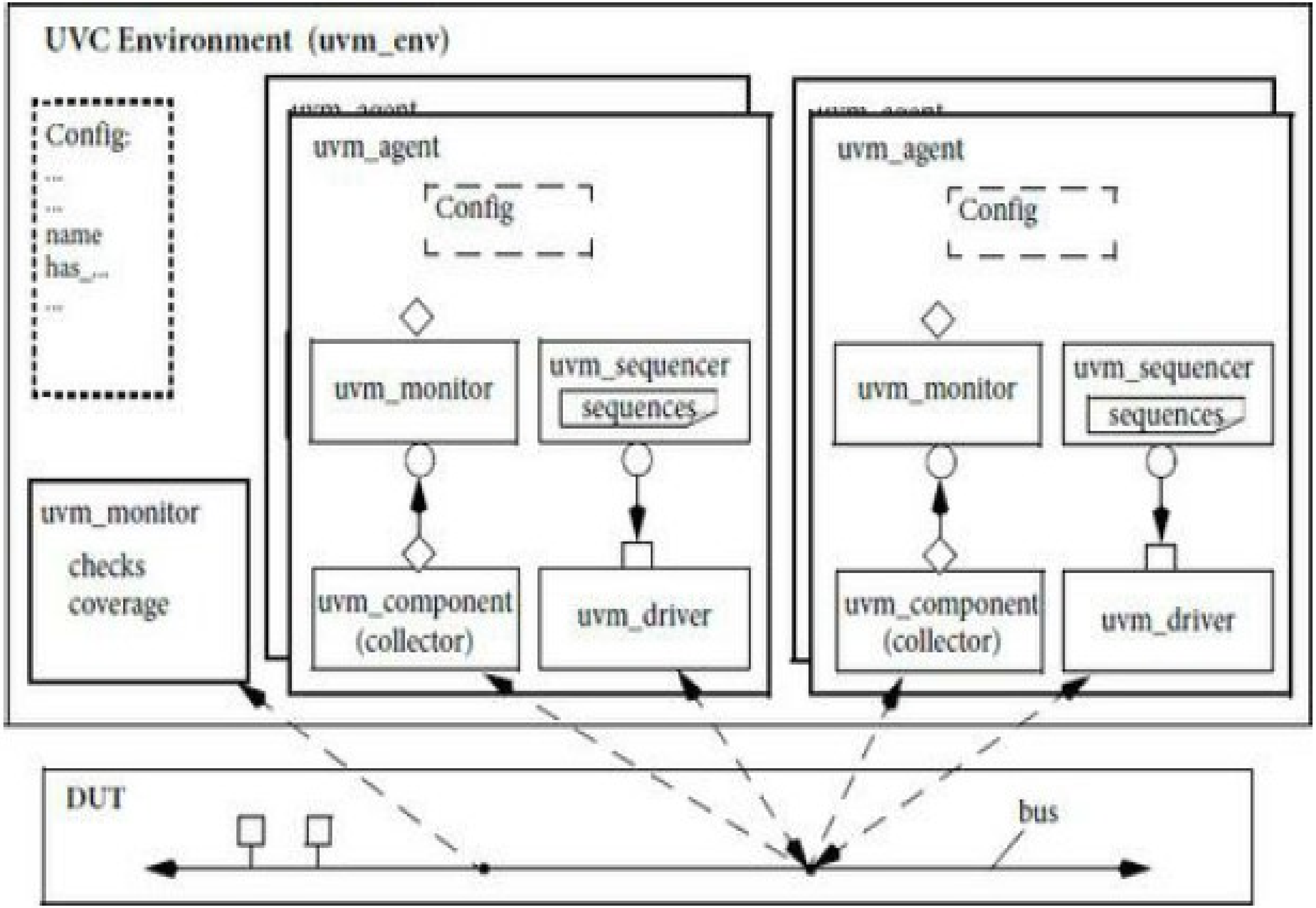
Env example to be revisited further



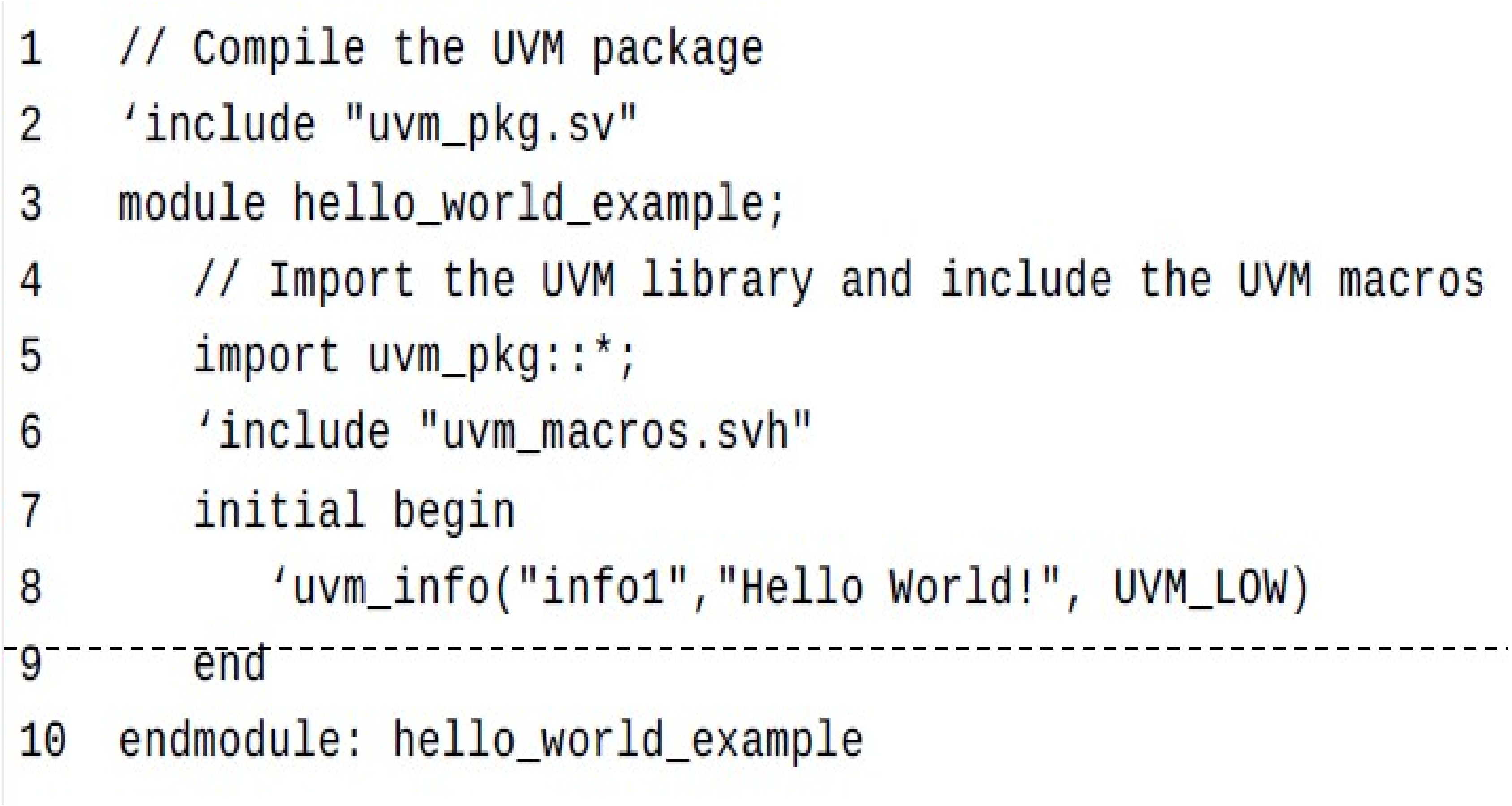
```
class env extends uvm_env;

my_component m_my_component;
my_param_component #(.ADDR_WIDTH(32), .DATA_WIDTH(32)) m_my_p_component;

// Constructor & registration macro left out

// Component and parameterized component create examples
function void build_phase( uvm_phase phase );
 m_my_component = my_component::type_id::create("m_my_component", this);
 m_my_p_component = my_param_component #(32, 32)::type_id::create
 ("m_my_p_component", this);
endfunction: build

task run_phase( uvm_phase phase );


 my_seq test_seq;
```



Note: Methodology classes are gray


UVC Environment (uvm_env)
Config:
...
...
name
has_...
...
uvm_monitor
checks
coverage
uvm_agent
uvm_agent
Config
uvm_monitor
uvm_sequencer
sequences
uvm_component
(collector)
uvm_driver
uvm_agent
uvm_agent
Config
uvm_monitor
uvm_sequencer
sequences
uvm_component
(collector)
uvm_driver
DUT
bus

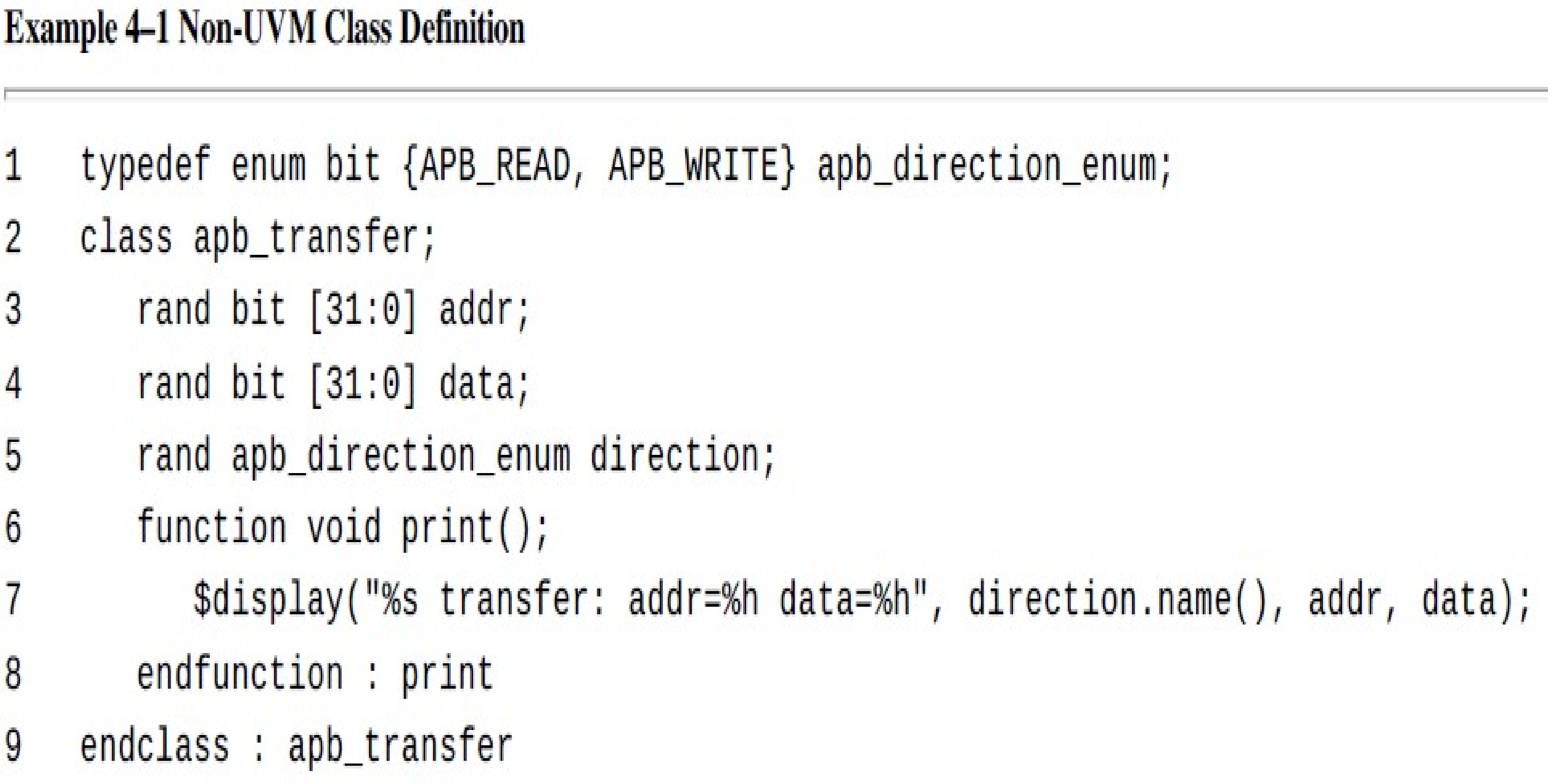
Hello world



```
// Compile the UVM package
‘include "uvm_pkg.sv"
module hello_world_example;
   // Import the UVM library and include the UVM macros
   import uvm_pkg::*;
   ‘include "uvm_macros.svh"
   initial begin
      ‘uvm_info("info1","Hello World!", UVM_LOW)
   end
endmodule: hello_world_example
```

- □ Lines 1-2: The comment is a reminder to compile the UVM library. The uvm_pkg.sv is the top UVM library file that includes the rest of the UVM files into a single SystemVerilog package.
- □ Line 5: When the library has been compiled, the user imports the package into any scopes that use the library features.
- □ Line 6: The UVM macros need to be included separately because they are compiler directives that do not survive multiple compilation steps.
- □ To avoid recompiling the entire library multiple times, they are included separately.
- □ Line 8: The `uvm_info macro is part of the UVM message capabilities that allow printing, formatting and controlling screen messages. In this case, we just print the message “Hello World!”

uvm_object Class (seq_item)

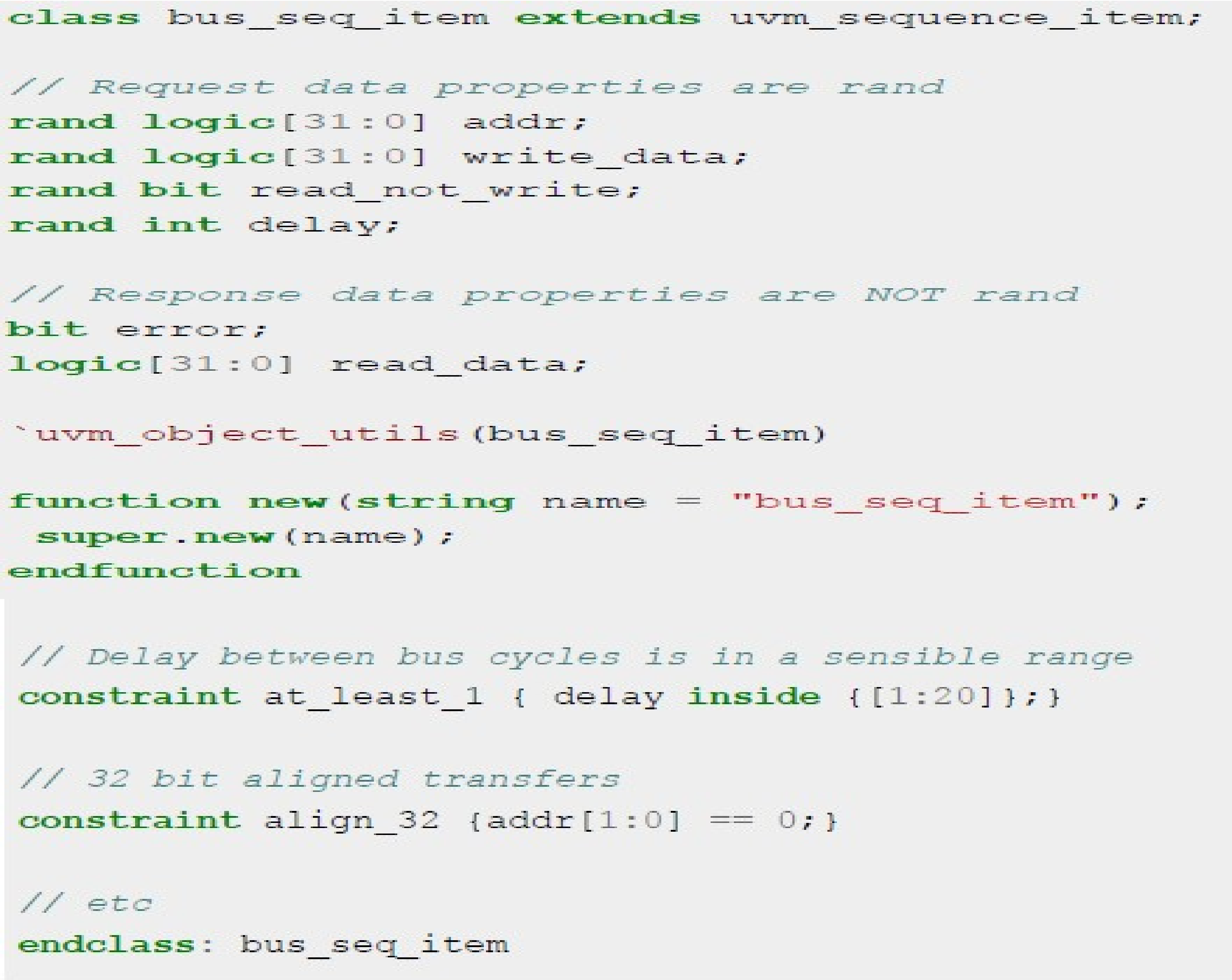


Example 4-1 Non-UVM Class Definition

```
typedef enum bit {APB_READ, APB_WRITE} apb_direction_enum;
class apb_transfer;
   rand bit [31:0] addr;
   rand bit [31:0] data;
   rand apb_direction_enum direction;
   function void print();
      $display("%s transfer: addr=%h data=%h", direction.name(), addr, data);
   endfunction : print
endclass : apb_transfer
```

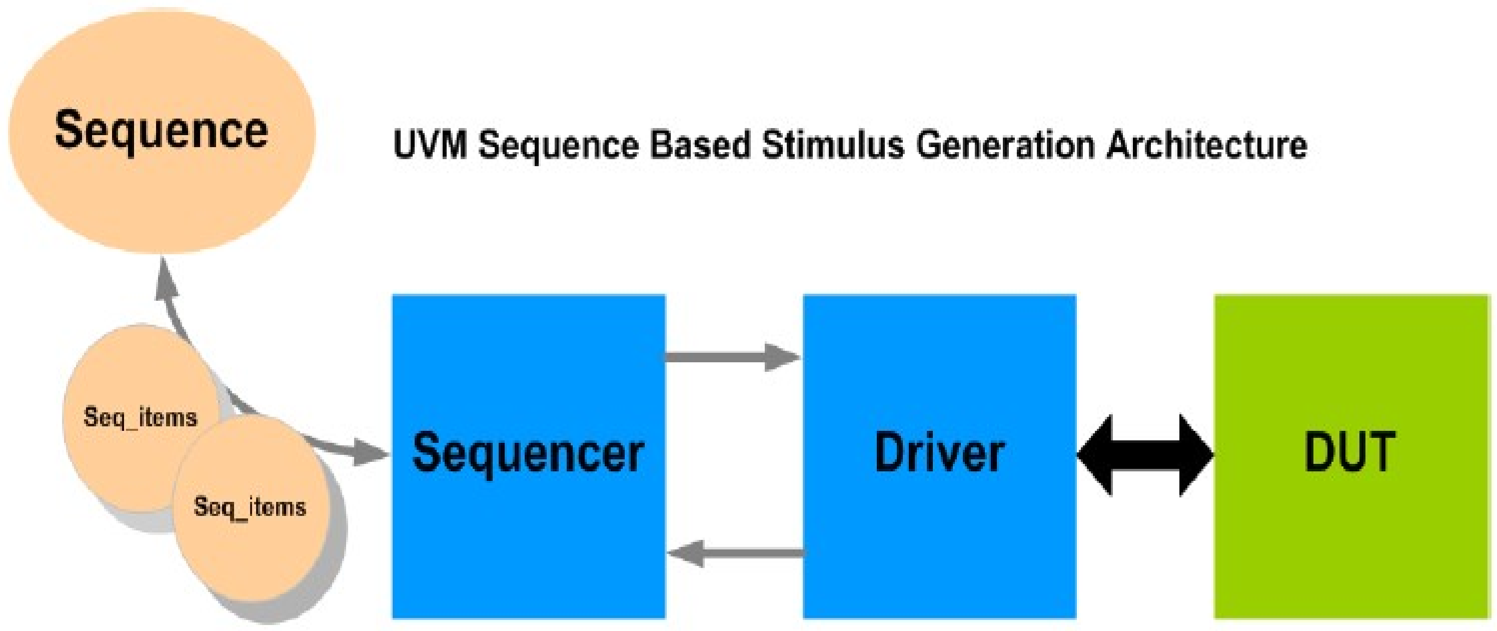
Example 4–2 APB Transfer Derived from uvm_object

```
typedef enum bit {APB_READ, APB_WRITE} apb_direction_enum;
class apb_transfer extends uvm_object;
   rand bit [31:0] addr;
   rand bit [31:0] data;
   rand apb_direction_enum direction;
   // Control field - does not translate into signal data
   rand int unsigned transmit_delay; //delay between transfers
   //UVM automation macros for data items
   ‘uvm_object_utils_begin(apb_transfer)
      ‘uvm_field_int(addr, UVM_DEFAULT)
      ‘uvm_field_int(data, UVM_DEFAULT)
      ‘uvm_field_enum(apb_direction_enum, direction, UVM_DEFAULT)
      ‘uvm_field_int(transmit_delay, UVM_DEFAULT | UVM_NOCOMPARE)
   ‘uvm_object_utils_end
   // Constructor - required UVM syntax
   function new (string name="apb_transfer");
      super.new(name);
   endfunction : new
endclass : apb_transfer
```



```
class bus_seq_item extends uvm_sequence_item;

// Request data properties are rand
rand logic[31:0] addr;
rand logic[31:0] write_data;
rand bit read_not_write;
rand int delay;

// Response data properties are NOT rand
bit error;
logic[31:0] read_data;

`uvm_object_utils(bus_seq_item)

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

// Delay between bus cycles is in a sensible range
constraint at_least_1 { delay inside {[1:20]};}

// 32 bit aligned transfers
constraint align_32 {addr[1:0] == 0;}

// etc
endclass: bus_seq_item
```

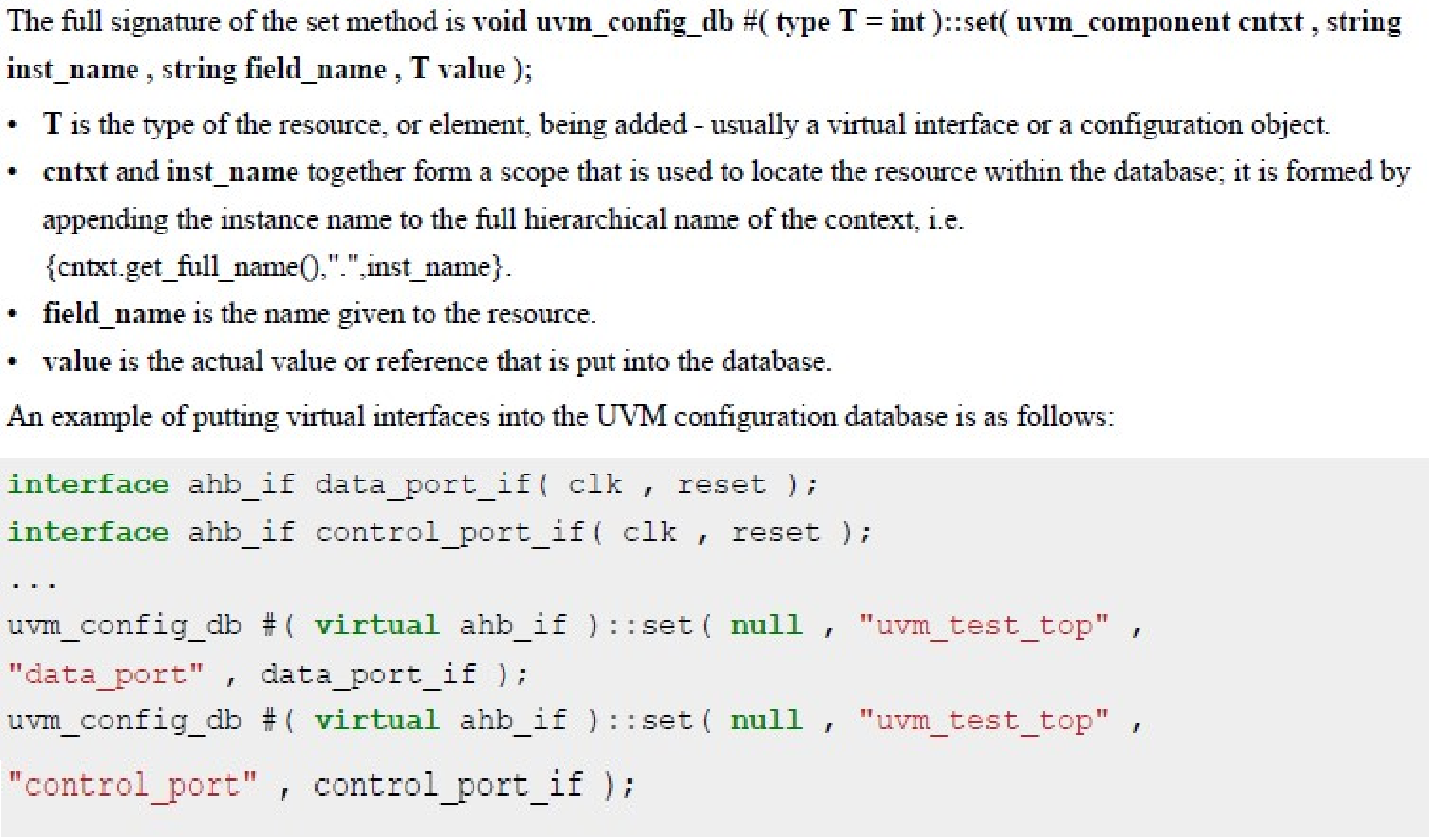
UVM Sequences



UVM Configuration Database (uvm_config_db)

- The uvm_config_db class is the recommended way to access the resource database. A resource is any piece of information that is shared between two or more components or objects.
- Use uvm_config_db::set to put information into the database and use uvm_config_db::get to retrieve information from the database.
- There are no limitations on the type parameter, which can be a class, a uvm_object, a built-in type like a bit, byte, or a virtual interface, etc.
- There are two typical uses of the uvm_config_db. The first is to pass virtual interfaces from the HDL/DUT domain to the test, and the second is to pass configuration objects down through the testbench hierarchy.

Set

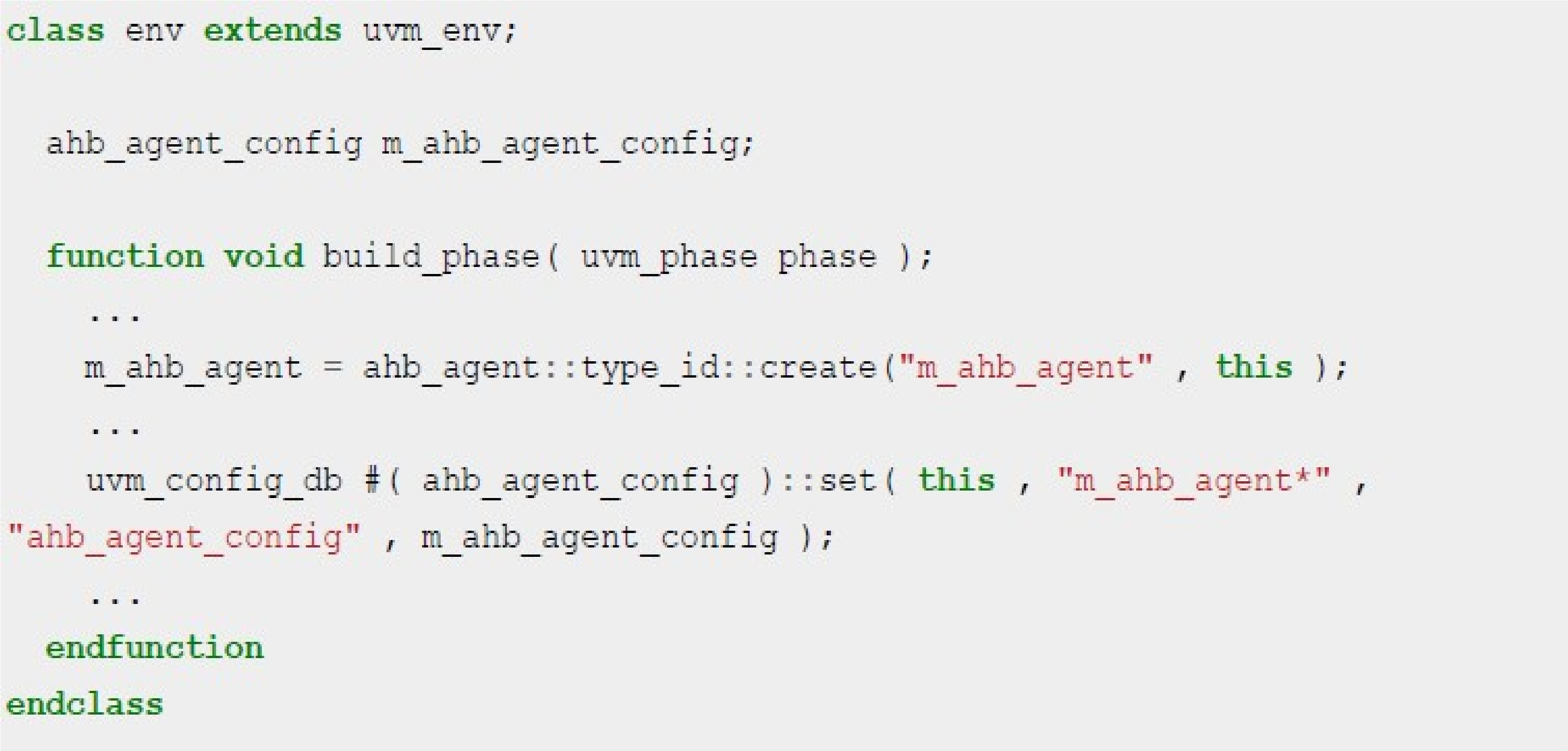


The full signature of the set method is **void uvm_config_db #(type T = int)::set(uvm_component cntxt , string inst_name , string field_name , T value);**

- **T** is the type of the resource, or element, being added - usually a virtual interface or a configuration object.
- **cntxt** and **inst_name** together form a scope that is used to locate the resource within the database; it is formed by appending the instance name to the full hierarchical name of the context, i.e. {cntxt.get_full_name(),".",inst_name}.
- **field_name** is the name given to the resource.
- **value** is the actual value or reference that is put into the database.

An example of putting virtual interfaces into the UVM configuration database is as follows:

```
interface ahb_if data_port_if( clk , reset );
interface ahb_if control_port_if( clk , reset );
...
uvm_config_db #( virtual ahb_if )::set( null , "uvm_test_top" ,
"data_port" , data_port_if );
uvm_config_db #( virtual ahb_if )::set( null , "uvm_test_top" ,
"control_port" , control_port_if );
```



```
class env extends uvm_env;

  ahb_agent_config m_ahb_agent_config;

  function void build_phase( uvm_phase phase );
    ...
    m_ahb_agent = ahb_agent::type_id::create("m_ahb_agent" , this );
    ...
    uvm_config_db #( ahb_agent_config )::set( this , "m_ahb_agent*" ,
"ahb_agent_config" , m_ahb_agent_config );
    ...
  endfunction
endclass
```

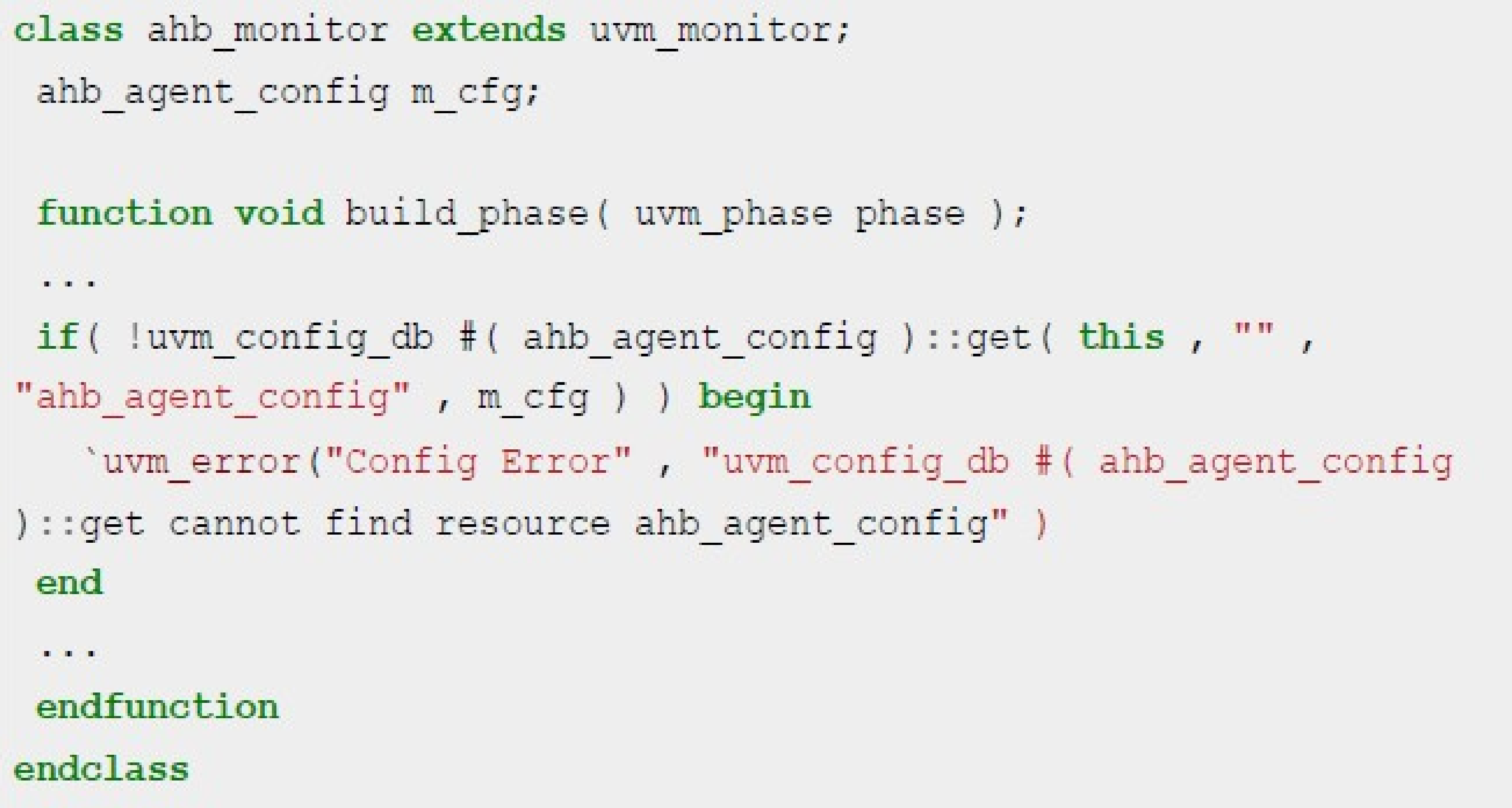
This code sets the configuration for the AHB agent and all its child components. Two things to note:

- Use "**this**" as the first argument to ensure that only *this* agent's configuration is set, and not of any other ahb_agent in the component hierarchy.
- Use "**m_ahb_agent***" to ensure that both the agent and its children are in the look-up scope. Without the '*' only the agent itself would be, and its driver, sequencer and monitor sub-components would be unable to access the configuration.

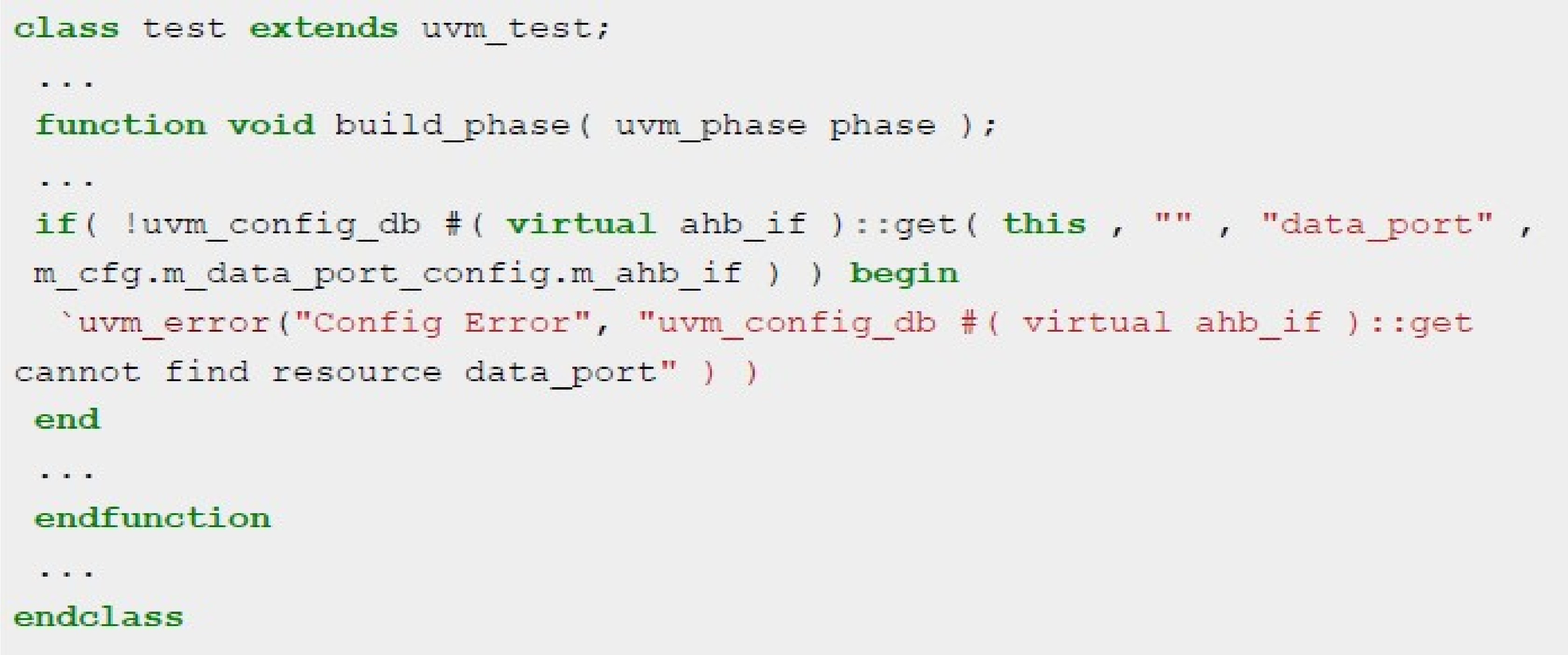
The get method

The full signature of the get method is **bit uvm_config_db #(type T = int) ::get(uvm_component cntxt , string inst_name , string field_name , ref T value);**

- **T** is the type of the resource, or element, being retrieved - usually a virtual interface or a configuration object.
- **cntxt** and **inst_name** together form a scope that is used to locate the resource within the database; it is formed by appending the instance name to the full hierarchical name of the context, i.e. {cntxt.get_full_name(),".",inst_name}.
- **field_name** is the name given to the resource.
- **value** holds the actual value or reference that is retrieved from the database; the get() call **returns** 1 if that retrieval succeeds, or else 0 indicating that no resource of this type and with this context and name exists in the database.

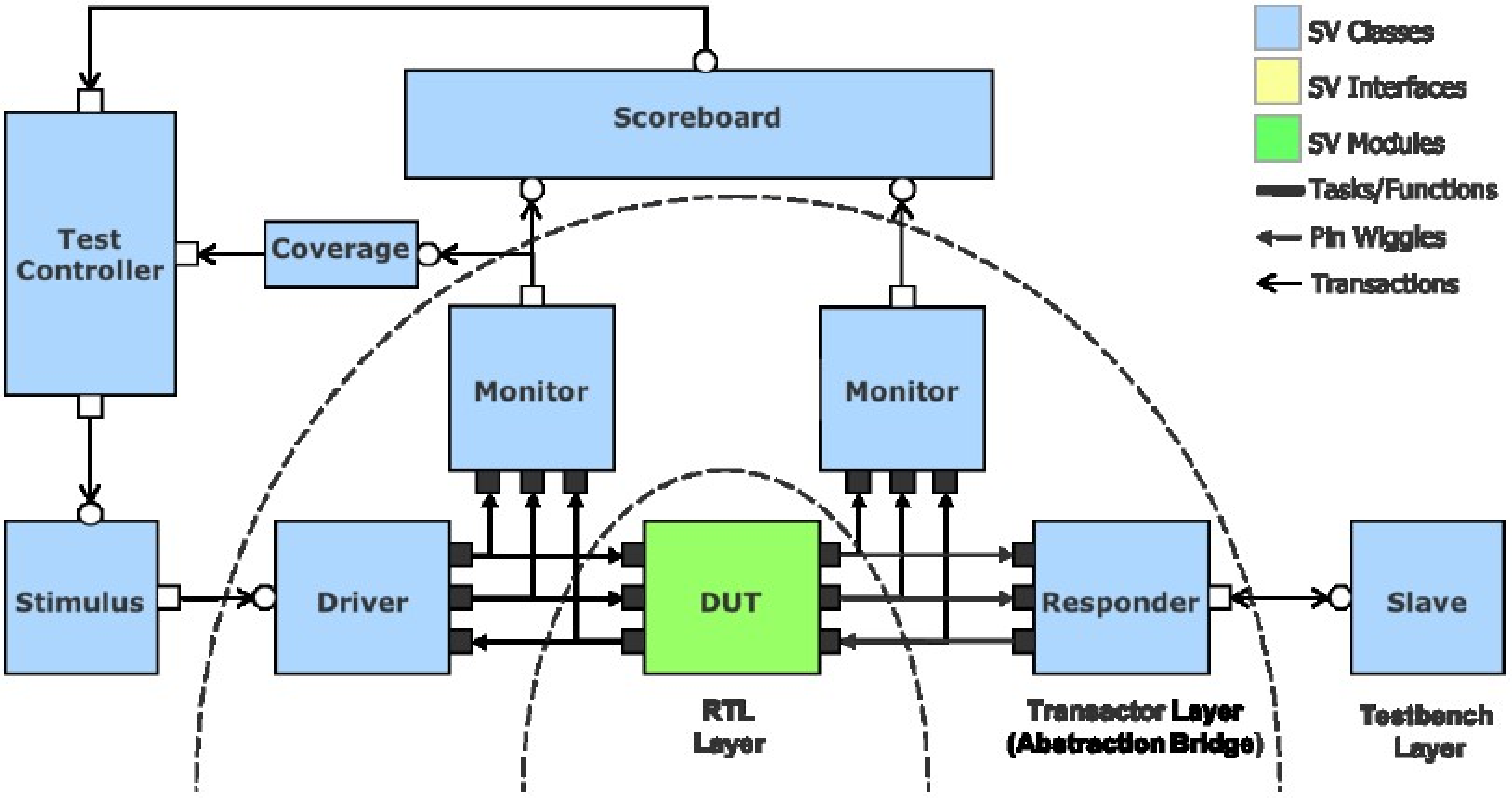


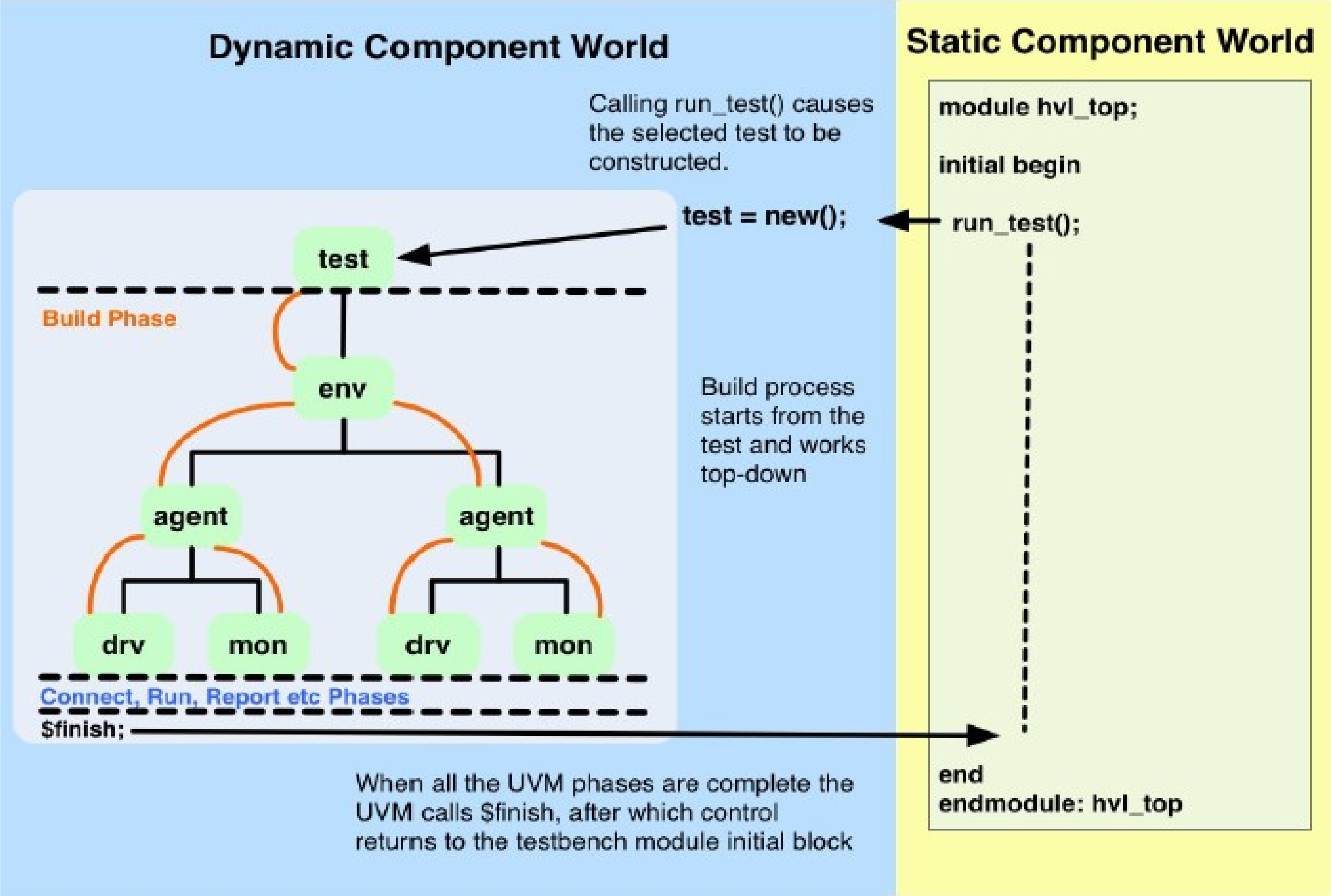
```
class ahb_monitor extends uvm_monitor;
 ahb_agent_config m_cfg;

 function void build_phase( uvm_phase phase );
 ...
 if( !uvm_config_db #( ahb_agent_config )::get( this , "" ,
"ahb_agent_config" , m_cfg ) ) begin
   `uvm_error("Config Error" , "uvm_config_db #( ahb_agent_config
)::get cannot find resource ahb_agent_config" )
 end
 ...
 endfunction
endclass
```



```
class test extends uvm_test;
 ...
 function void build_phase( uvm_phase phase );
 ...
 if( !uvm_config_db #( virtual ahb_if )::get( this , "" , "data_port" ,
 m_cfg.m_data_port_config.m_ahb_if ) ) begin
  `uvm_error("Config Error", "uvm_config_db #( virtual ahb_if )::get
cannot find resource data_port" ) )
 end
 ...
 endfunction
 ...
endclass
```

Testbench Architecture





Calling run_test() causes the selected test to be constructed.

```
test = new();
```

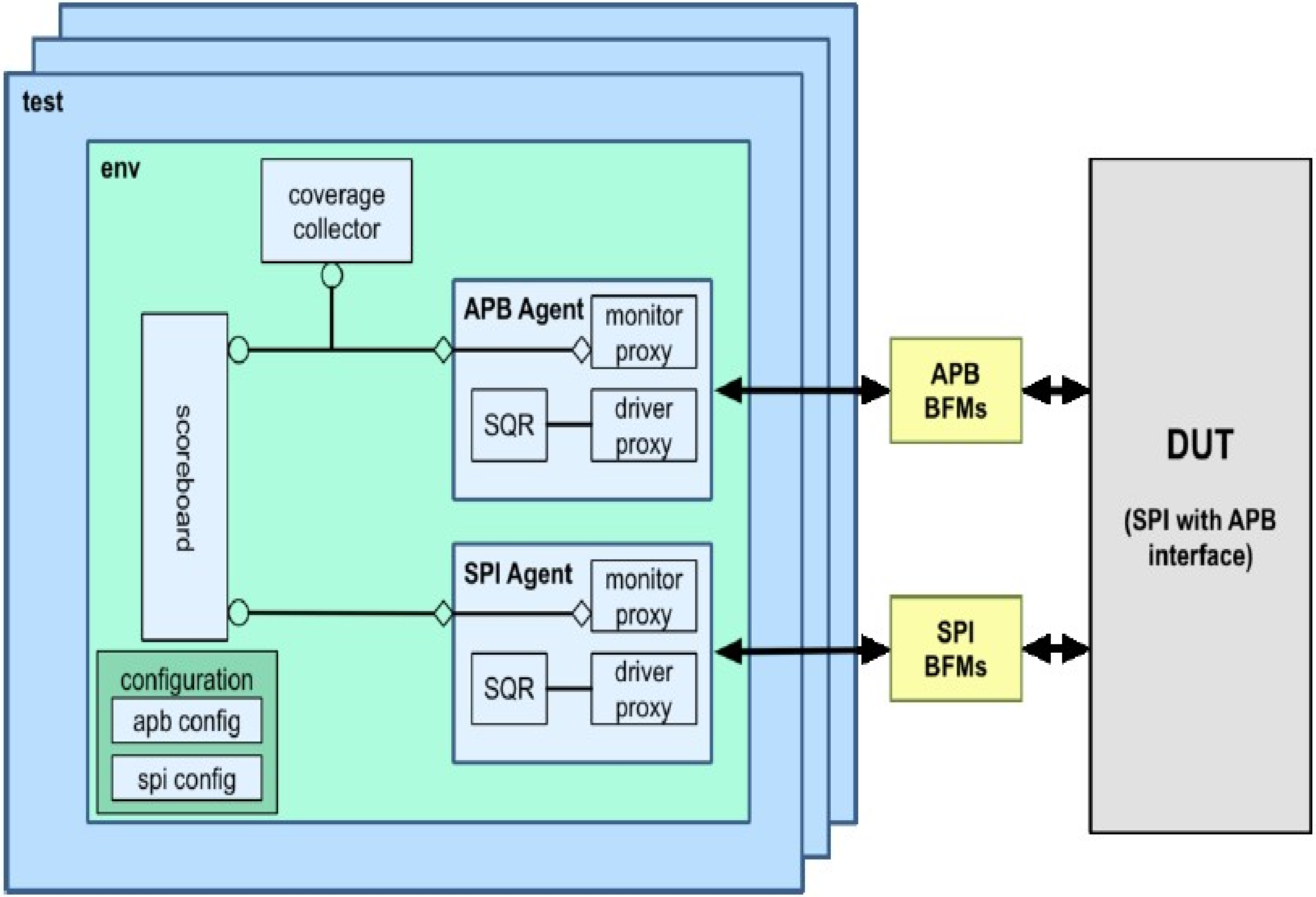
Build process starts from the test and works top-down

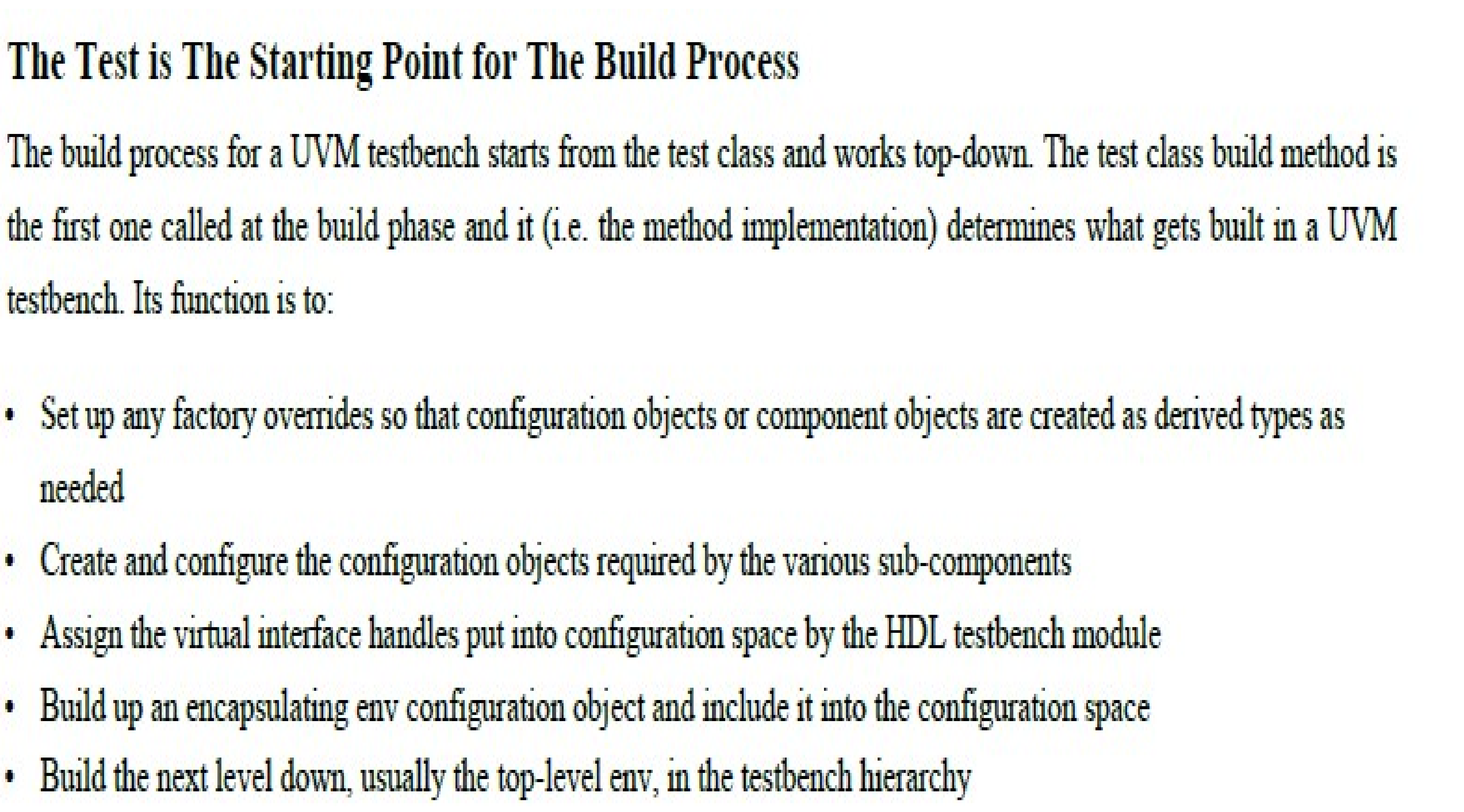
```
$finish;
```

When all the UVM phases are complete the UVM calls $finish, after which control returns to the testbench module initial block

```
module hvl_top;

initial begin

 run_test();

end
endmodule: hvl_top
```

The UVM Build flow in the context of the testbench

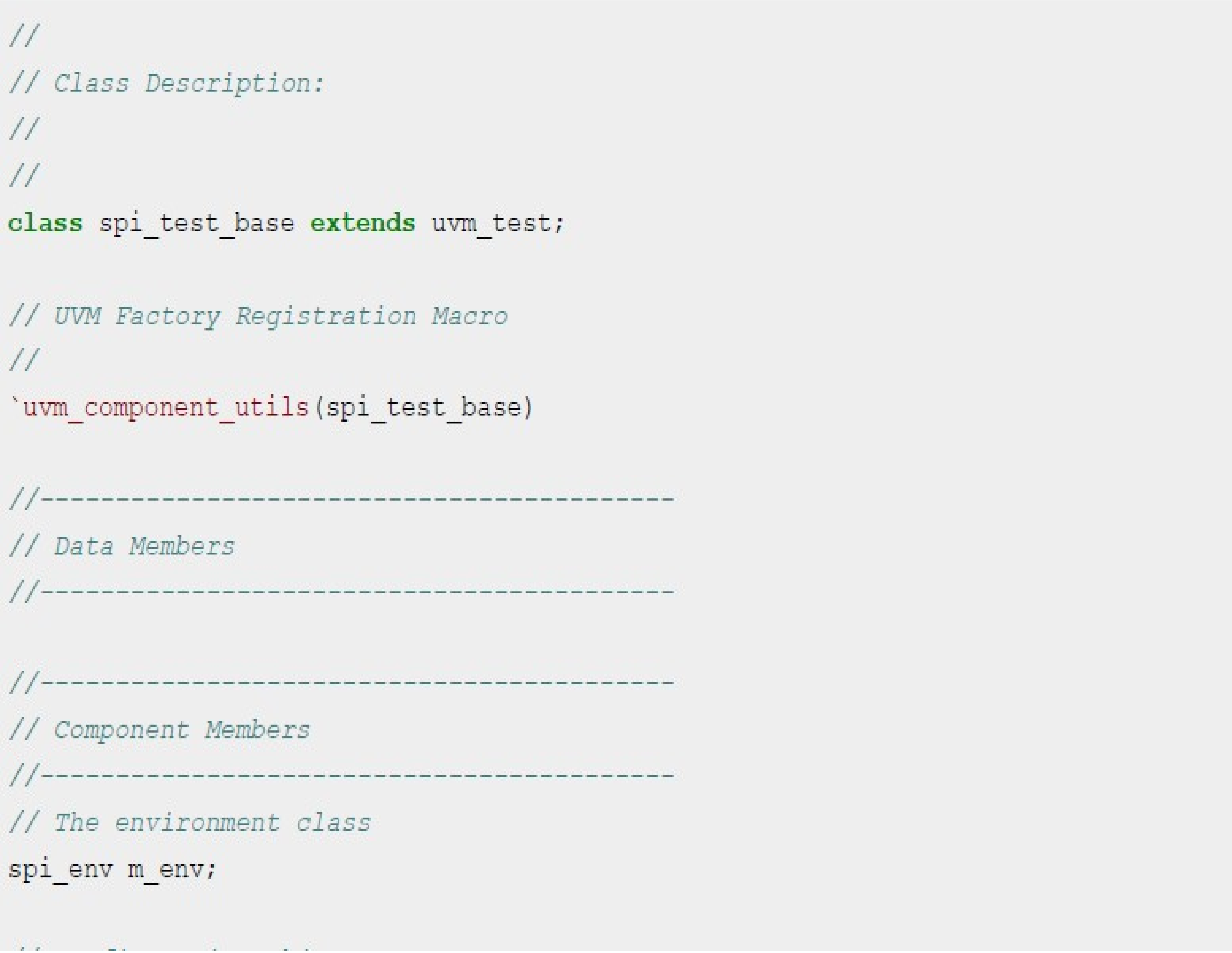

test
env
coverage collector
scoreboard
APB Agent
monitor proxy
SQR
driver proxy
SPI Agent
monitor proxy
SQR
driver proxy
configuration
apb config
spi config
APB BFMs
SPI BFMs
DUT
(SPI with APB interface)



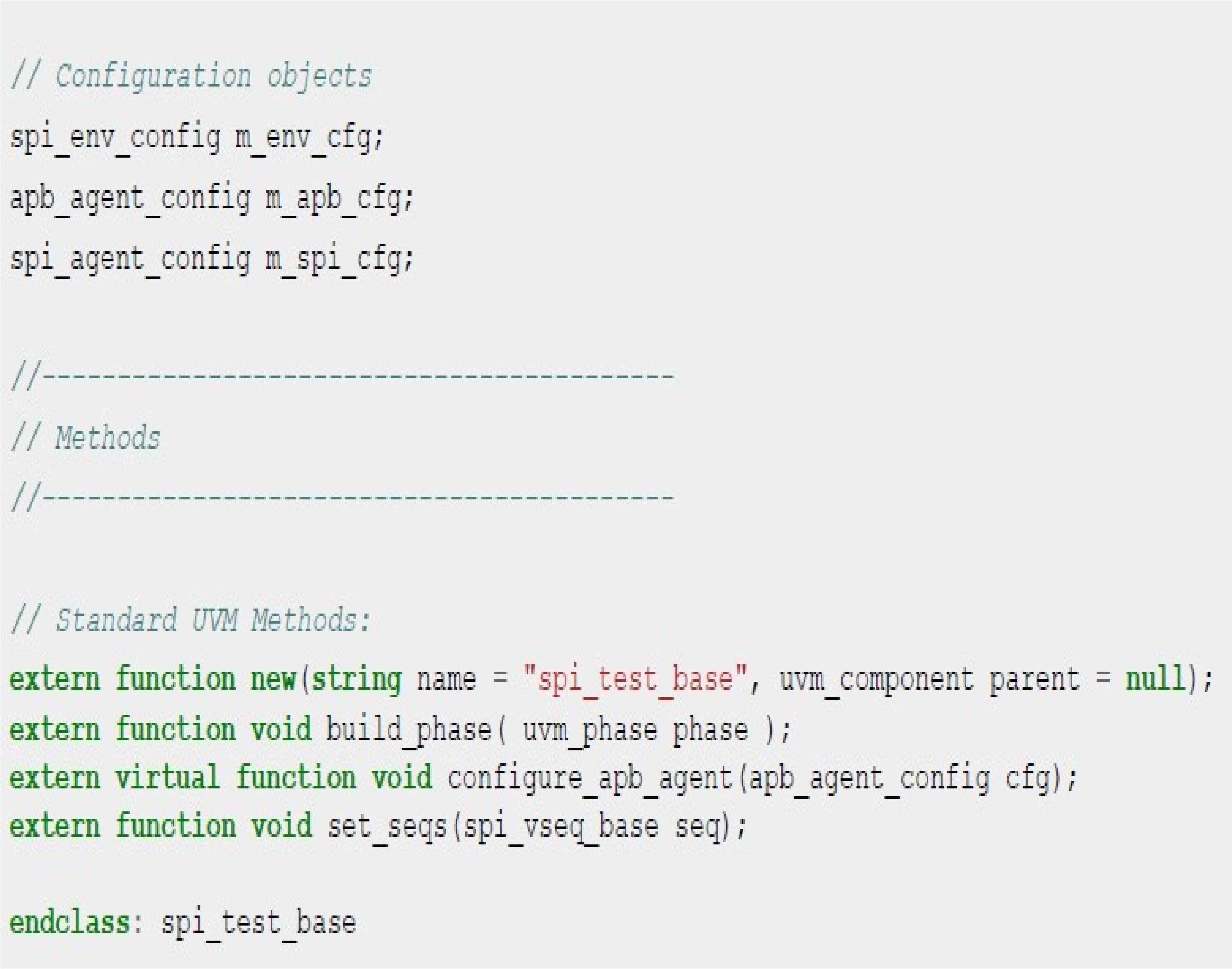
The Test is The Starting Point for The Build Process

The build process for a UVM testbench starts from the test class and works top-down. The test class build method is the first one called at the build phase and it (i.e. the method implementation) determines what gets built in a UVM testbench. Its function is to:

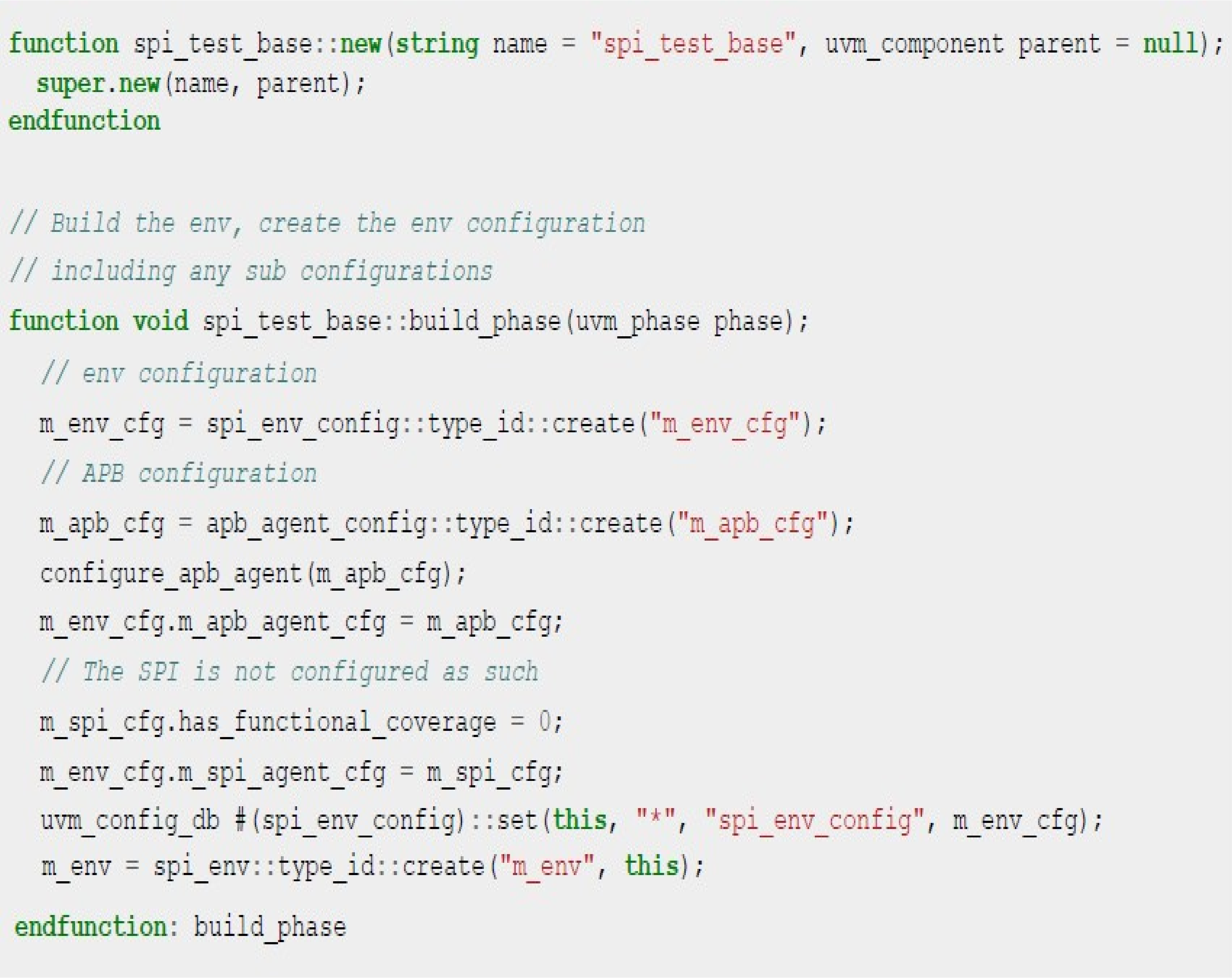
- Set up any factory overrides so that configuration objects or component objects are created as derived types as needed
- Create and configure the configuration objects required by the various sub-components
- Assign the virtual interface handles put into configuration space by the HDL testbench module
- Build up an encapsulating env configuration object and include it into the configuration space
- Build the next level down, usually the top-level env, in the testbench hierarchy



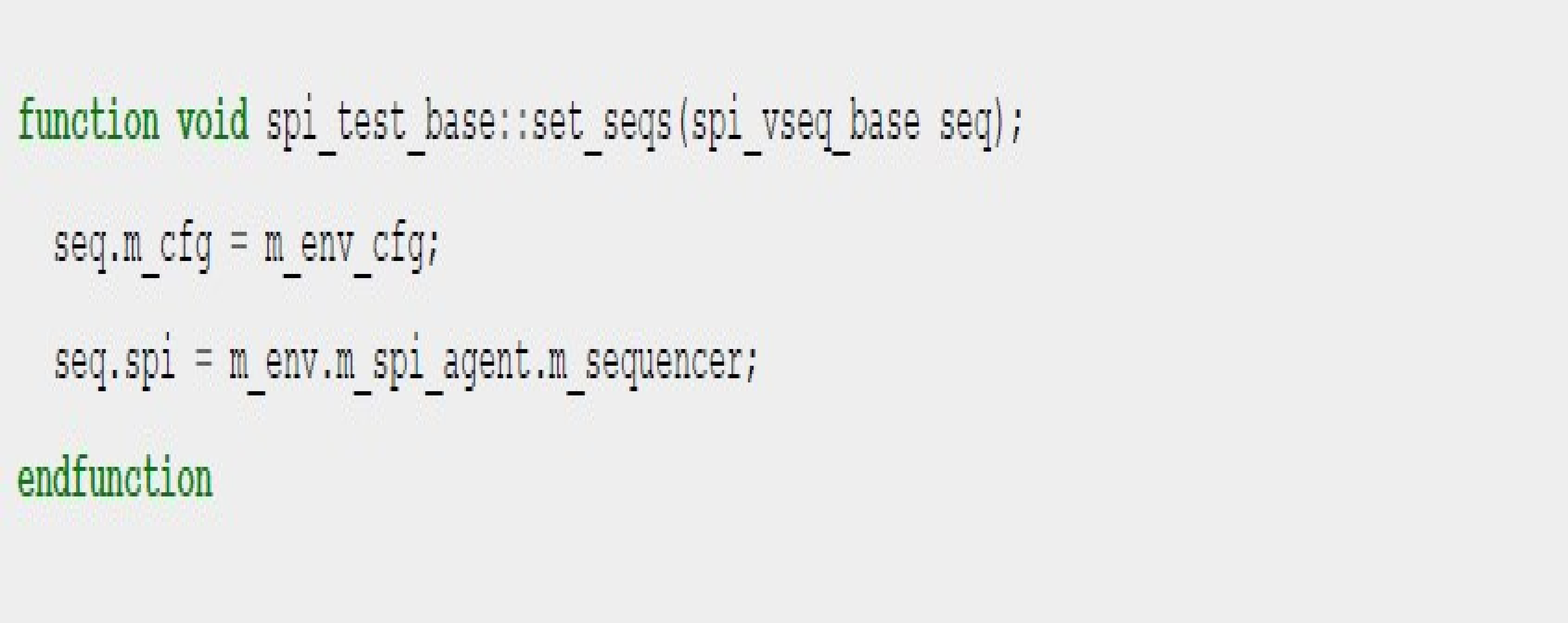
```
//
// Class Description:
//
//
class spi_test_base extends uvm_test;

// UVM Factory Registration Macro
//
`uvm_component_utils(spi_test_base)

//------------------------------------------
// Data Members
//------------------------------------------

//------------------------------------------
// Component Members
//------------------------------------------
// The environment class
spi_env m_env;
```



```
// Configuration objects
spi_env_config m_env_cfg;
apb_agent_config m_apb_cfg;
spi_agent_config m_spi_cfg;

//------------------------------------------
// Methods
//------------------------------------------

// Standard UVM Methods:
extern function new(string name = "spi_test_base", uvm_component parent = null);
extern function void build_phase( uvm_phase phase );
extern virtual function void configure_apb_agent(apb_agent_config cfg);
extern function void set_seqs(spi_vseq_base seq);

endclass: spi_test_base
```

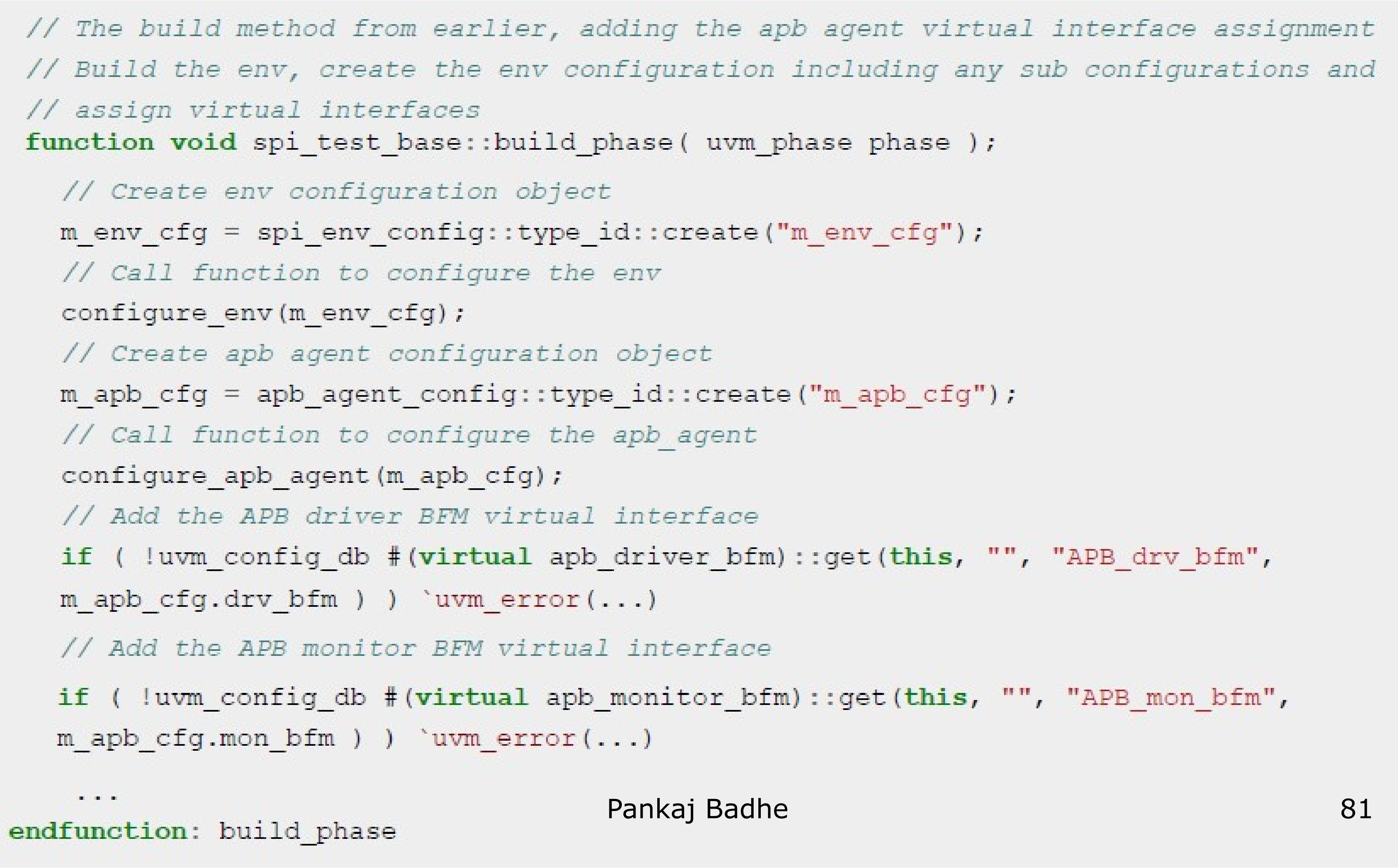


```
function spi_test_base::new(string name = "spi_test_base", uvm_component parent = null);
  super.new(name, parent);
endfunction


// Build the env, create the env configuration
// including any sub configurations
function void spi_test_base::build_phase(uvm_phase phase);
  // env configuration
  m_env_cfg = spi_env_config::type_id::create("m_env_cfg");
  // APB configuration
  m_apb_cfg = apb_agent_config::type_id::create("m_apb_cfg");
  configure_apb_agent(m_apb_cfg);
  m_env_cfg.m_apb_agent_cfg = m_apb_cfg;
  // The SPI is not configured as such
  m_spi_cfg.has_functional_coverage = 0;
  m_env_cfg.m_spi_agent_cfg = m_spi_cfg;
  uvm_config_db #(spi_env_config)::set(this, "*", "spi_env_config", m_env_cfg);
  m_env = spi_env::type_id::create("m_env", this);
endfunction: build_phase
```

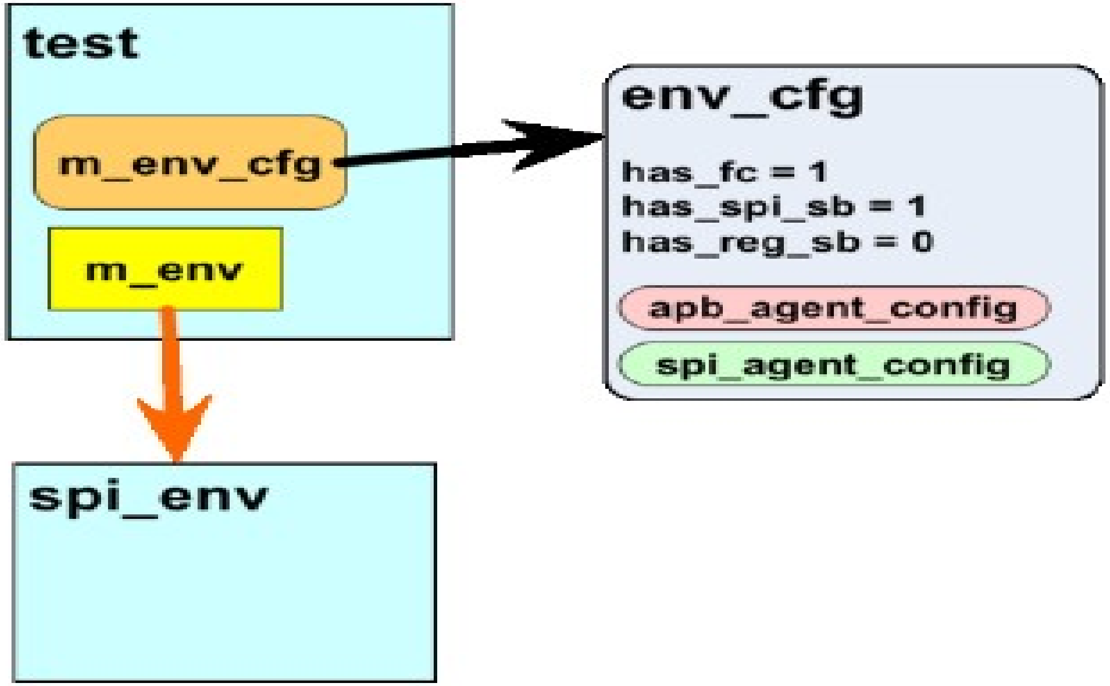


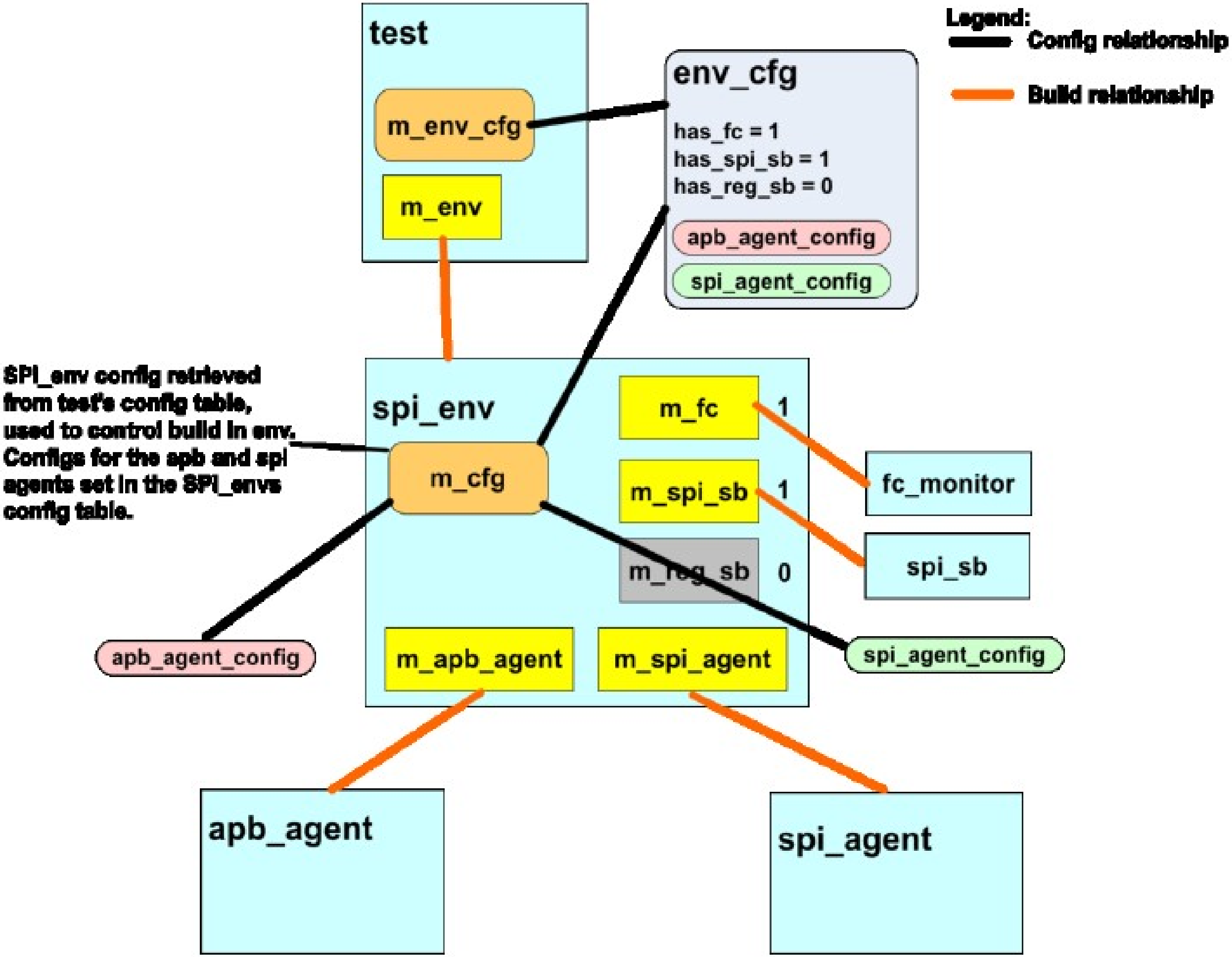
```
function void spi_test_base::set_seqs(spi_vseq_base seq);
  seq.m_cfg = m_env_cfg;
  seq.spi = m_env.m_spi_agent.m_sequencer;
endfunction
```

Assigning Virtual Interfaces From The Configuration Space

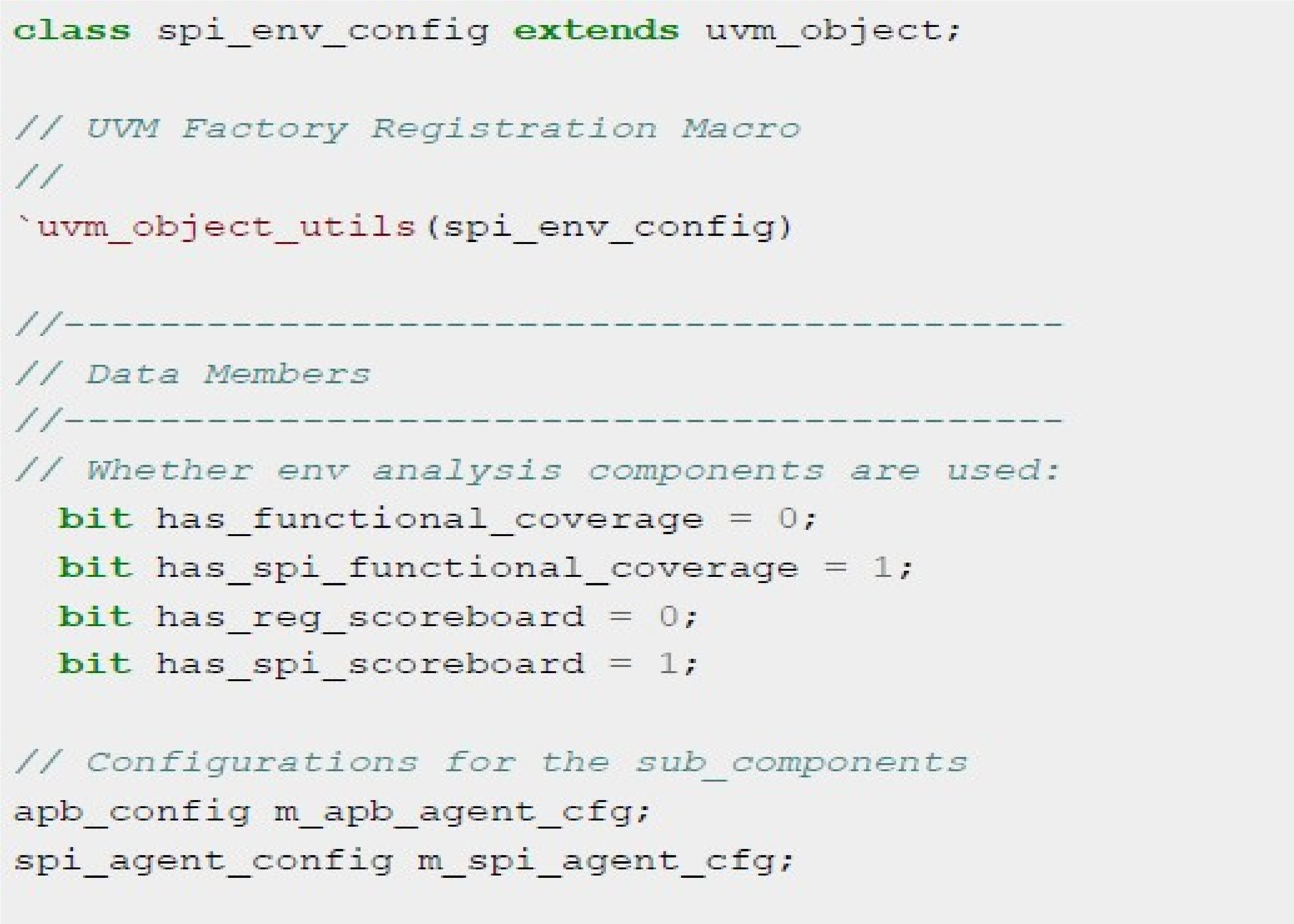


```
// The build method from earlier, adding the apb agent virtual interface assignment
// Build the env, create the env configuration including any sub configurations and
// assign virtual interfaces
function void spi_test_base::build_phase( uvm_phase phase );
  // Create env configuration object
  m_env_cfg = spi_env_config::type_id::create("m_env_cfg");
  // Call function to configure the env
  configure_env(m_env_cfg);
  // Create apb agent configuration object
  m_apb_cfg = apb_agent_config::type_id::create("m_apb_cfg");
  // Call function to configure the apb_agent
  configure_apb_agent(m_apb_cfg);
  // Add the APB driver BFM virtual interface
  if ( !uvm_config_db #(virtual apb_driver_bfm)::get(this, "", "APB_drv_bfm",
  m_apb_cfg.drv_bfm ) ) `uvm_error(...)
  // Add the APB monitor BFM virtual interface
  if ( !uvm_config_db #(virtual apb_monitor_bfm)::get(this, "", "APB_mon_bfm",
  m_apb_cfg.mon_bfm ) ) `uvm_error(...)
   ...
endfunction: build_phase
```

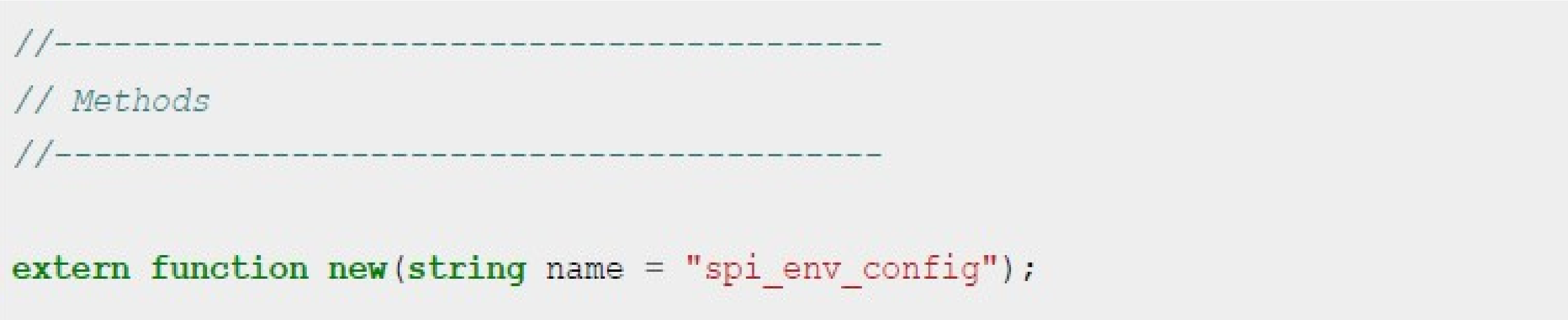

test
m_env_cfg
m_env
env_cfg
has_fc = 1
has_spi_sb = 1
has_reg_sb = 0
apb_agent_config
spi_agent_config
spi_env



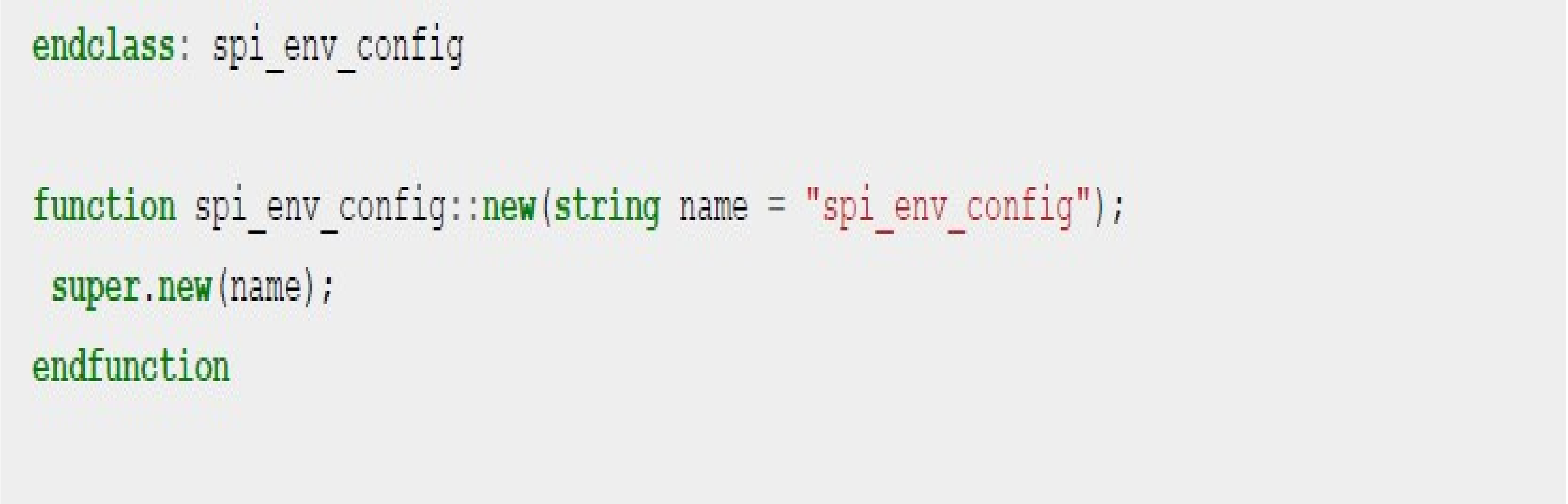
Testbench Build Process – Step 2 – At the end of spi_envs build() method



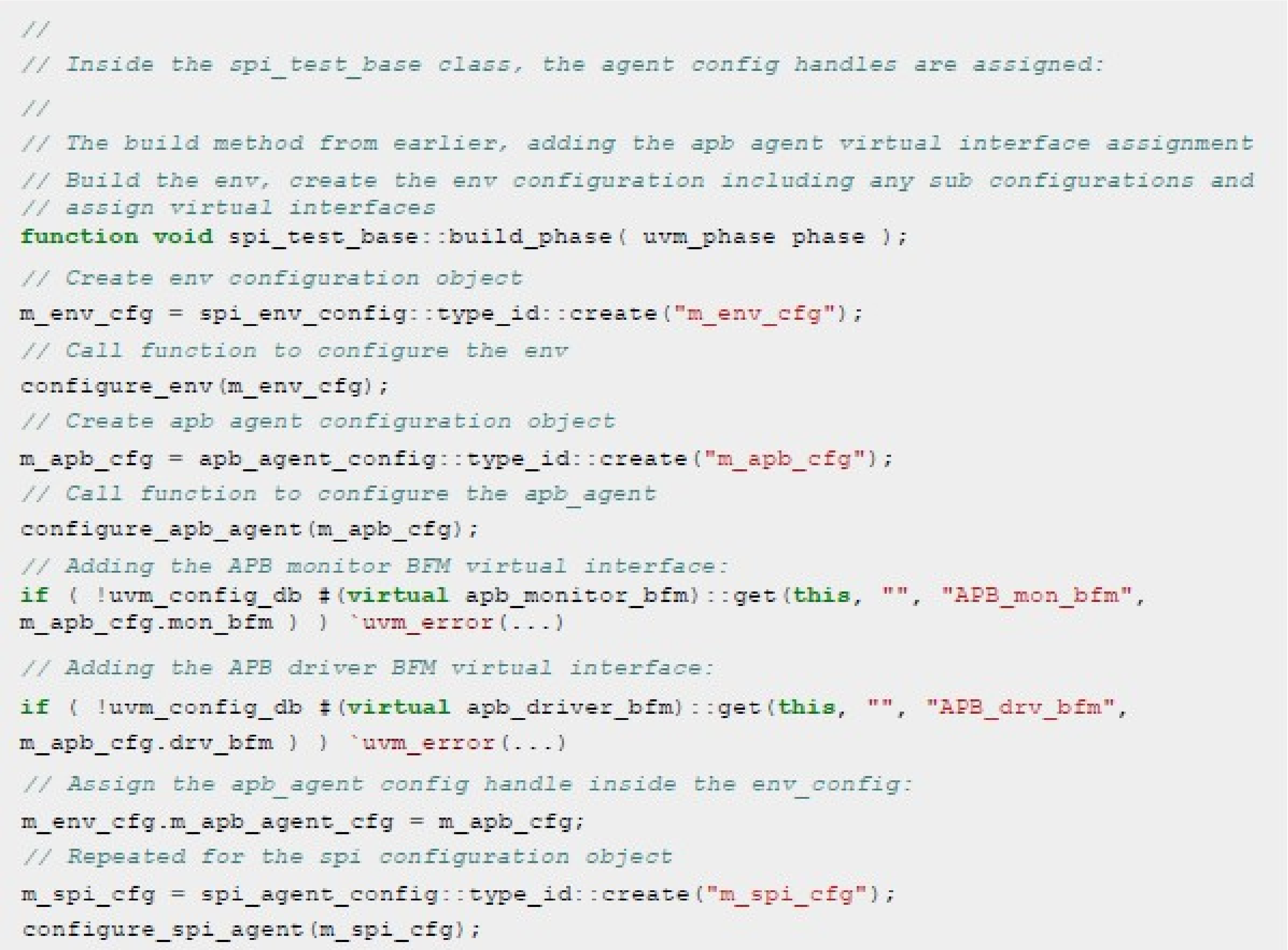
```
class spi_env_config extends uvm_object;

// UVM Factory Registration Macro
//
`uvm_object_utils(spi_env_config)

//------------------------------------------
// Data Members
//------------------------------------------
// Whether env analysis components are used:
  bit has_functional_coverage = 0;
  bit has_spi_functional_coverage = 1;
  bit has_reg_scoreboard = 0;
  bit has_spi_scoreboard = 1;

// Configurations for the sub_components
apb_config m_apb_agent_cfg;
spi_agent_config m_spi_agent_cfg;
```



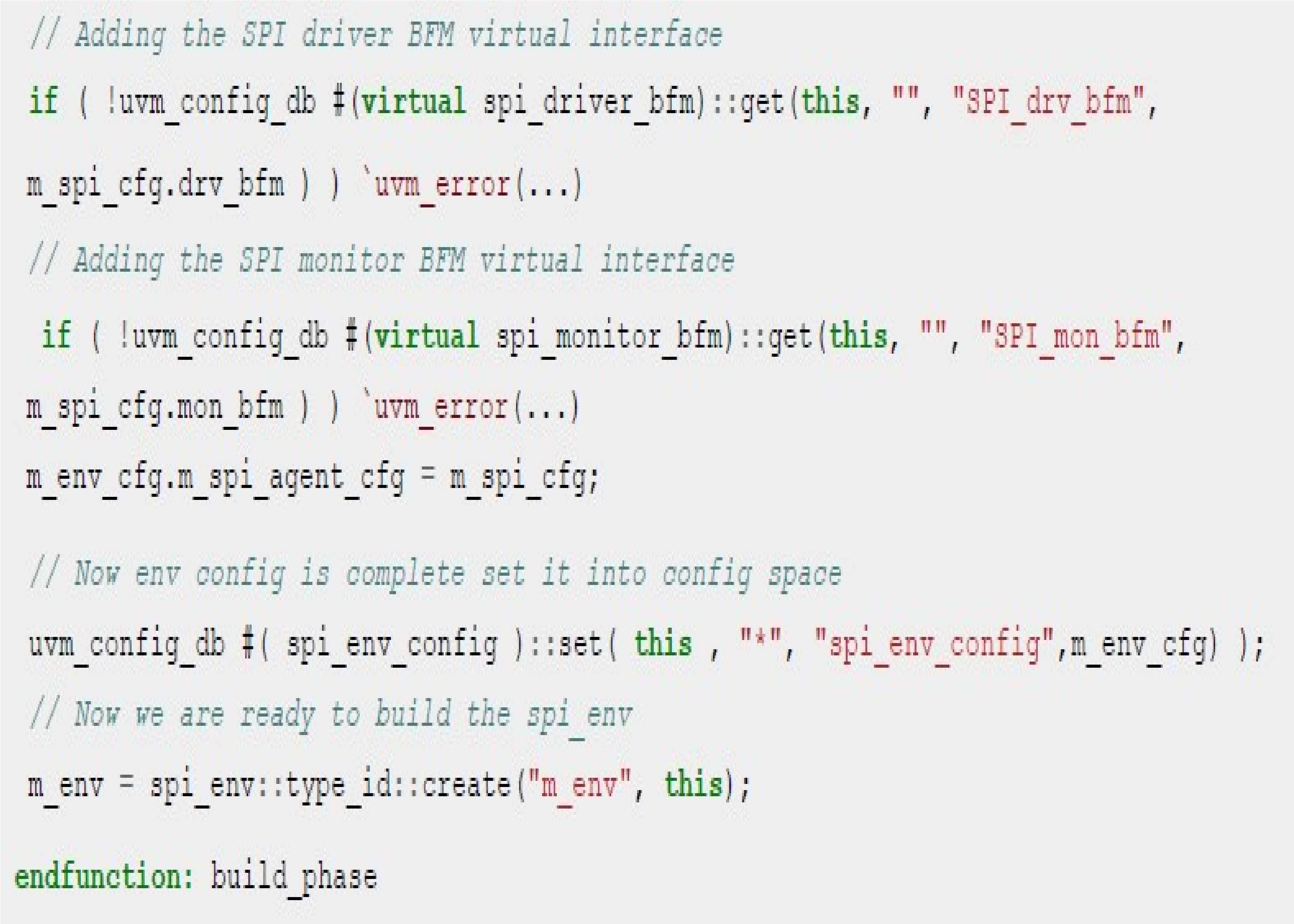
```
//------------------------------------------
// Methods
//------------------------------------------

extern function new(string name = "spi_env_config");
```



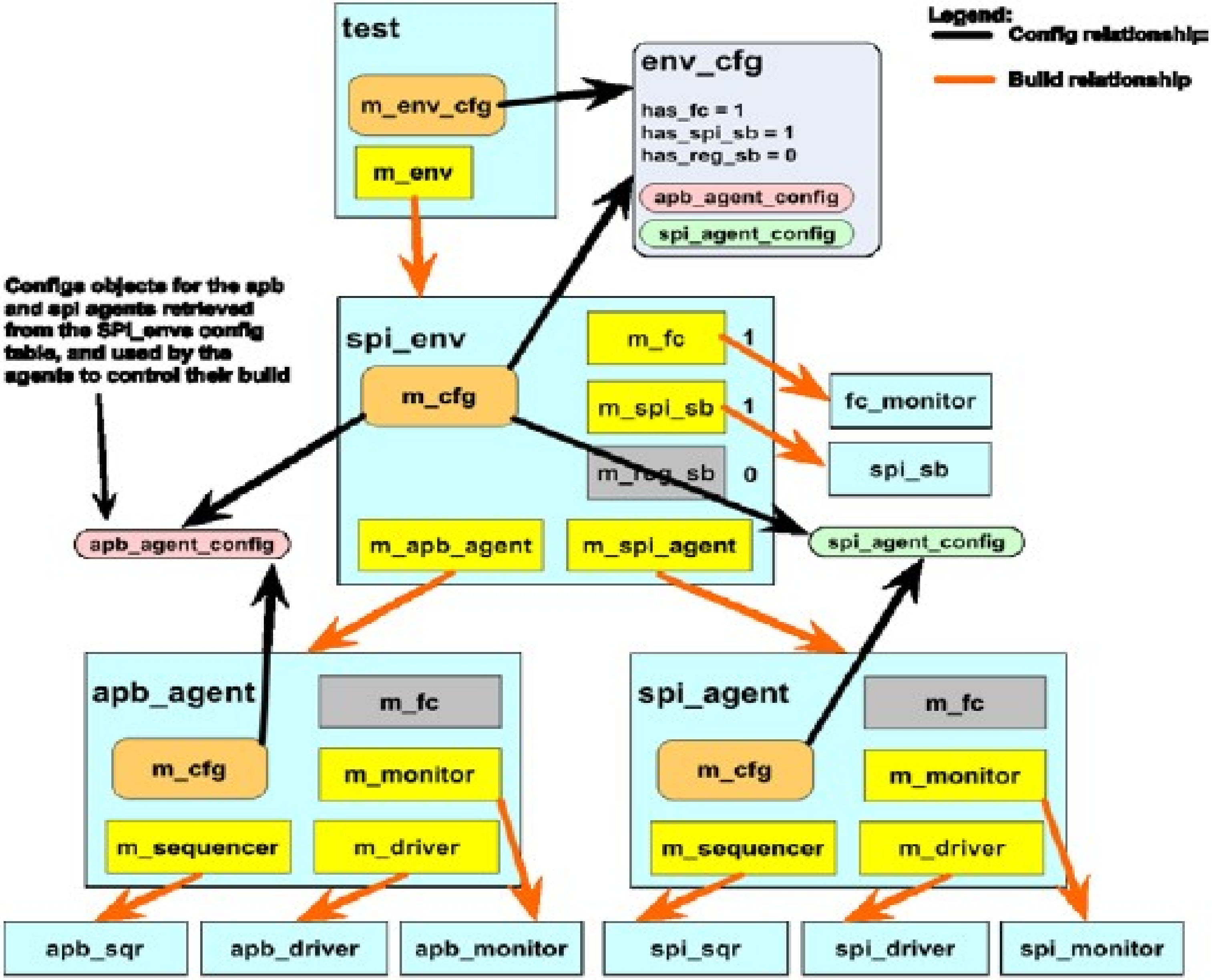
```
endclass: spi_env_config


function spi_env_config::new(string name = "spi_env_config");
 super.new(name);
endfunction
```



```
//
// Inside the spi_test_base class, the agent config handles are assigned:
//
// The build method from earlier, adding the apb agent virtual interface assignment
// Build the env, create the env configuration including any sub configurations and
// assign virtual interfaces
function void spi_test_base::build_phase( uvm_phase phase );
// Create env configuration object
m_env_cfg = spi_env_config::type_id::create("m_env_cfg");
// Call function to configure the env
configure_env(m_env_cfg);
// Create apb agent configuration object
m_apb_cfg = apb_agent_config::type_id::create("m_apb_cfg");
// Call function to configure the apb_agent
configure_apb_agent(m_apb_cfg);
// Adding the APB monitor BFM virtual interface:
if ( !uvm_config_db #(virtual apb_monitor_bfm)::get(this, "", "APB_mon_bfm",
m_apb_cfg.mon_bfm ) ) `uvm_error(...)
// Adding the APB driver BFM virtual interface:
if ( !uvm_config_db #(virtual apb_driver_bfm)::get(this, "", "APB_drv_bfm",
m_apb_cfg.drv_bfm ) ) `uvm_error(...)
// Assign the apb_agent config handle inside the env_config:
m_env_cfg.m_apb_agent_cfg = m_apb_cfg;
// Repeated for the spi configuration object
m_spi_cfg = spi_agent_config::type_id::create("m_spi_cfg");
configure_spi_agent(m_spi_cfg);
```

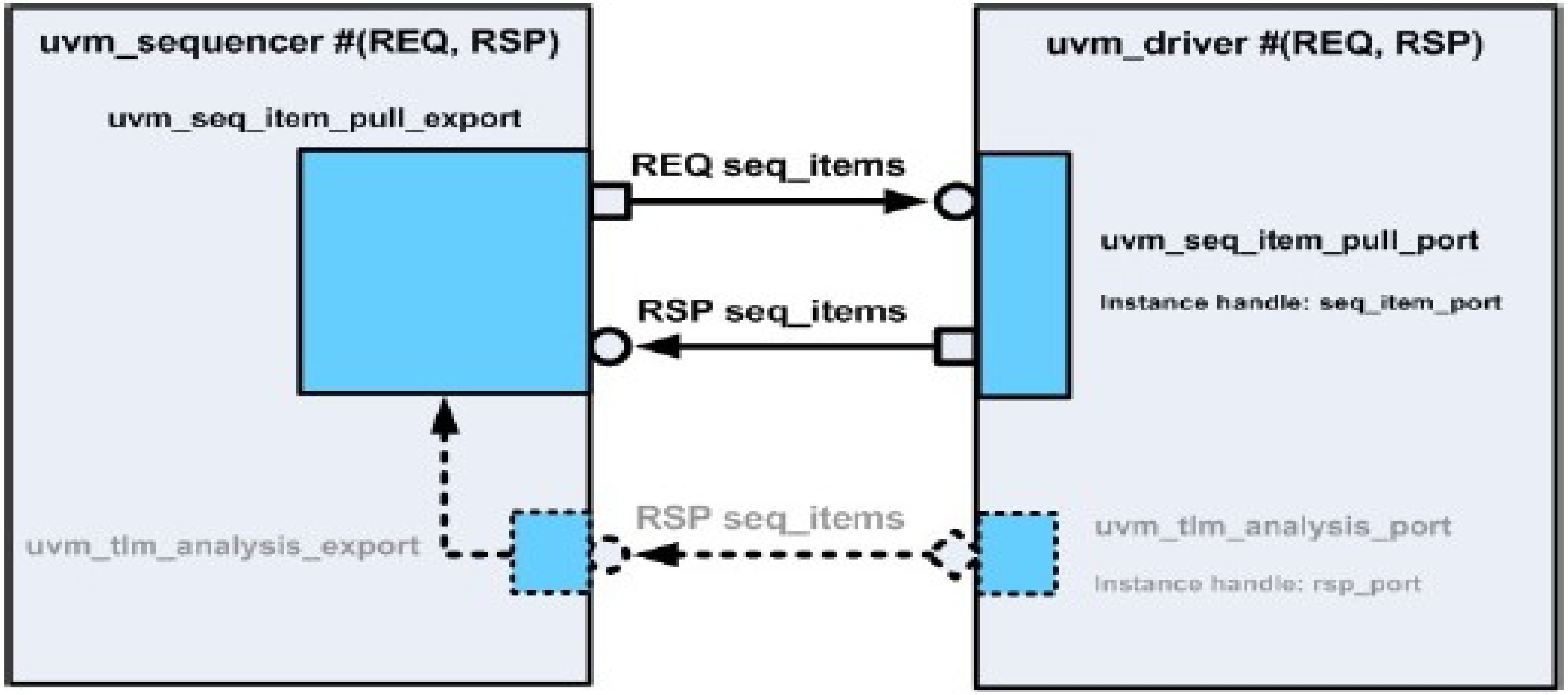


```
 // Adding the SPI driver BFM virtual interface
 if ( !uvm_config_db #(virtual spi_driver_bfm)::get(this, "", "SPI_drv_bfm",
 m_spi_cfg.drv_bfm ) ) `uvm_error(...)
 // Adding the SPI monitor BFM virtual interface
  if ( !uvm_config_db #(virtual spi_monitor_bfm)::get(this, "", "SPI_mon_bfm",
 m_spi_cfg.mon_bfm ) ) `uvm_error(...)
 m_env_cfg.m_spi_agent_cfg = m_spi_cfg;

 // Now env config is complete set it into config space
 uvm_config_db #( spi_env_config )::set( this , "*", "spi_env_config",m_env_cfg) );
 // Now we are ready to build the spi_env
 m_env = spi_env::type_id::create("m_env", this);

endfunction: build_phase
```

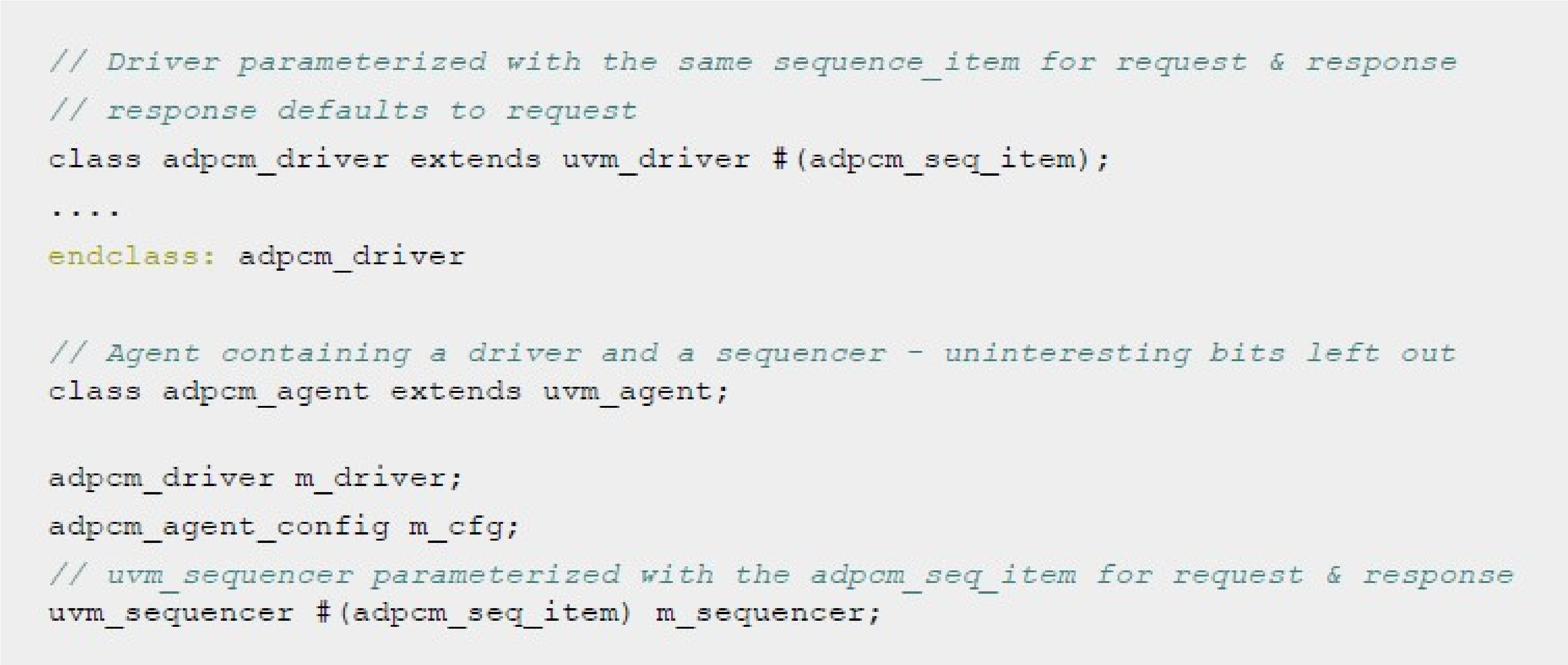


Testbench Build Process – Stage 3 - End of Agent build() method

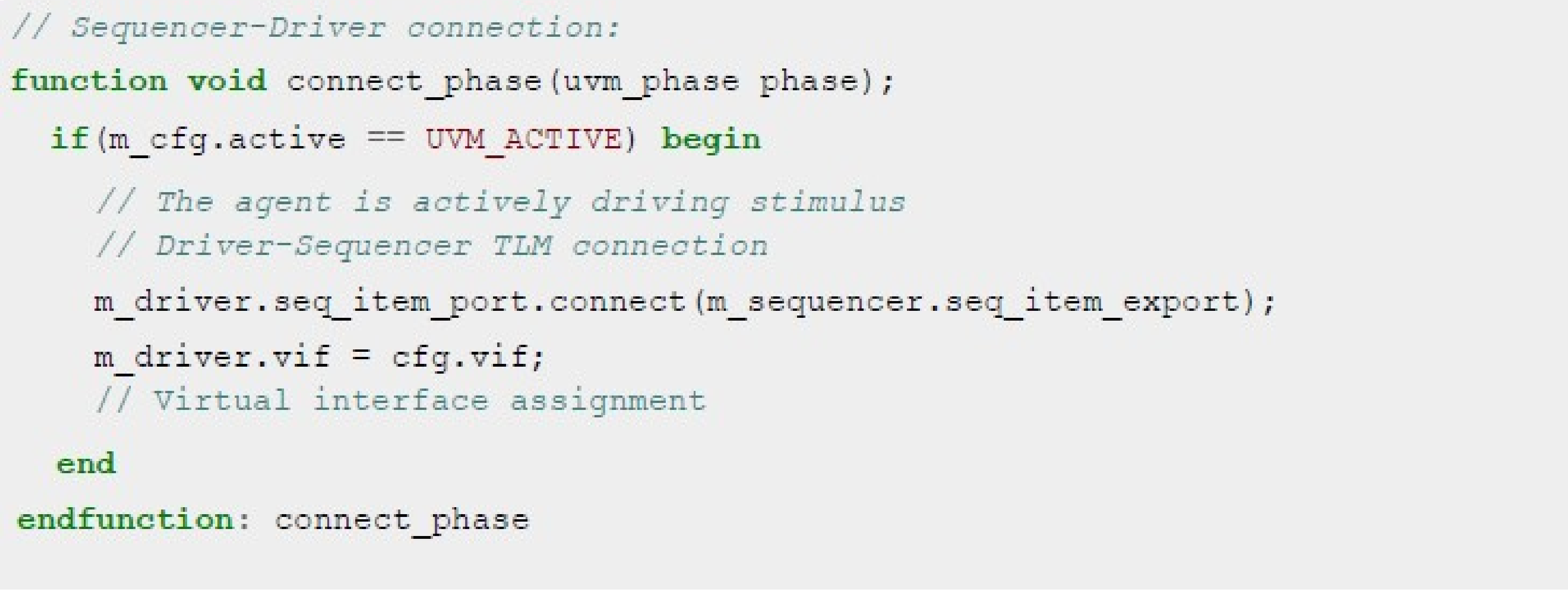
Connecting the Sequencer and Driver



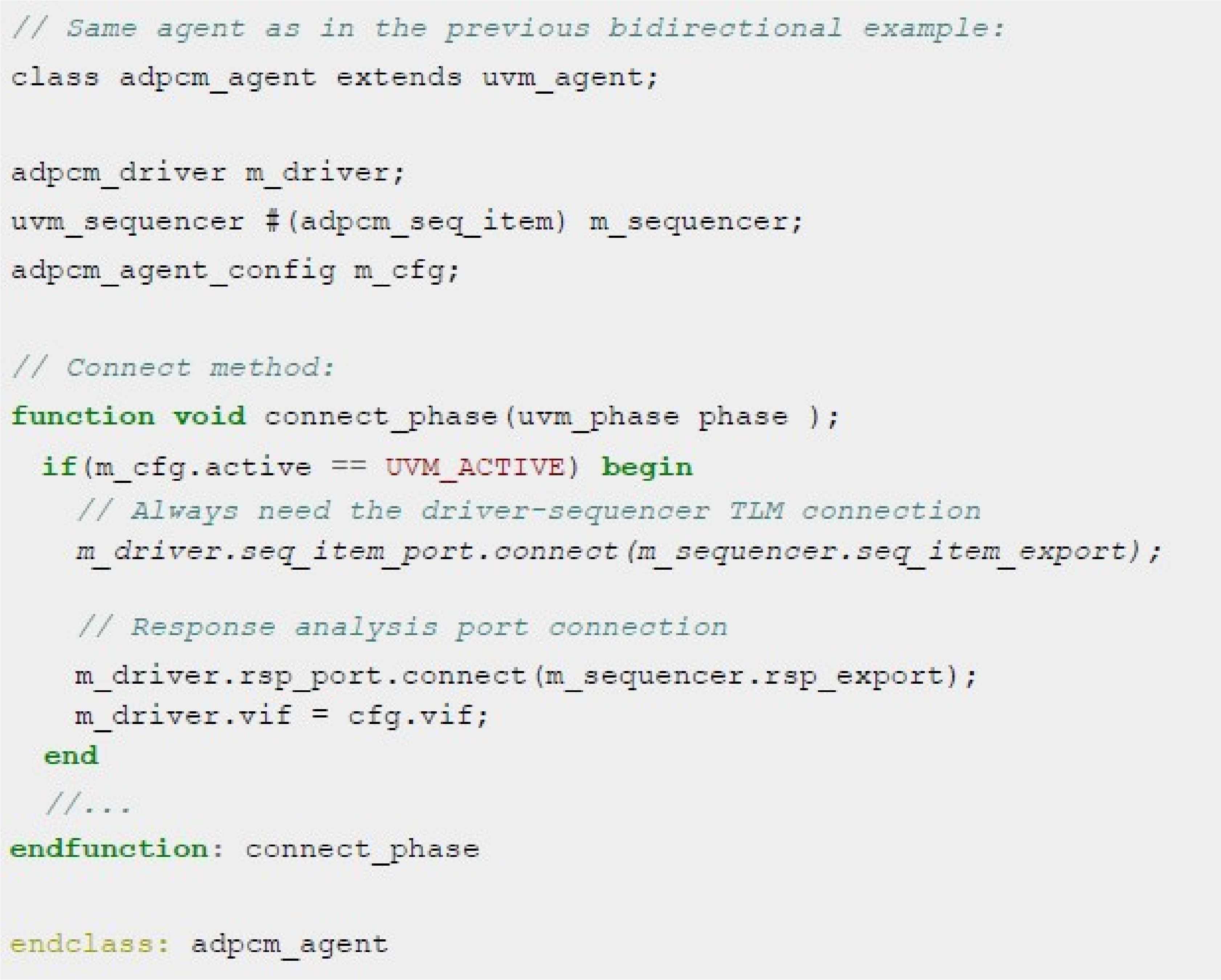
Sequencer-Driver Connections



```
// Driver parameterized with the same sequence_item for request & response
// response defaults to request
class adpcm_driver extends uvm_driver #(adpcm_seq_item);
....
endclass: adpcm_driver

// Agent containing a driver and a sequencer - uninteresting bits left out
class adpcm_agent extends uvm_agent;

adpcm_driver m_driver;
adpcm_agent_config m_cfg;
// uvm_sequencer parameterized with the adpcm_seq_item for request & response
uvm_sequencer #(adpcm_seq_item) m_sequencer;
```



```
// Sequencer-Driver connection:
function void connect_phase(uvm_phase phase);
  if(m_cfg.active == UVM_ACTIVE) begin
    // The agent is actively driving stimulus
    // Driver-Sequencer TLM connection
    m_driver.seq_item_port.connect(m_sequencer.seq_item_export);
    m_driver.vif = cfg.vif;
    // Virtual interface assignment
  end
endfunction: connect_phase
```



```
// Same agent as in the previous bidirectional example:
class adpcm_agent extends uvm_agent;

adpcm_driver m_driver;
uvm_sequencer #(adpcm_seq_item) m_sequencer;
adpcm_agent_config m_cfg;

// Connect method:
function void connect_phase(uvm_phase phase );
  if(m_cfg.active == UVM_ACTIVE) begin
    // Always need the driver-sequencer TLM connection
    m_driver.seq_item_port.connect(m_sequencer.seq_item_export);

    // Response analysis port connection
    m_driver.rsp_port.connect(m_sequencer.rsp_export);
    m_driver.vif = cfg.vif;
  end
  //...
endfunction: connect_phase

endclass: adpcm_agent
```