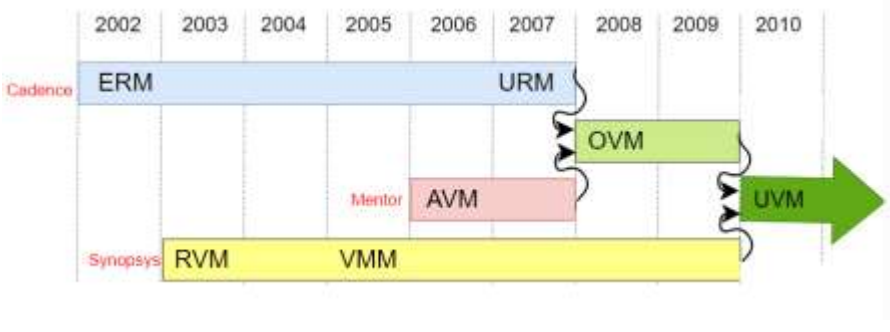
Lab 9: UVM

1. Introduction

What is UVM?

UVM stands for Universal Verification Methodology. UVM is a standardized class based transaction level methodology (**TLM**). UVM is built on top of the SystemVerilog language and provides a framework for creating modular, reusable testbench components that can be easily integrated into the design verification process. It also includes a set of guidelines and best practices for developing testbenches, as well as a methodology for running simulations and analyzing results.

Let's see how UVM came to being. There were three competing methodologies in the industry. Open Verification Methodology (OVM) from Mentor, Universal Reuse Methodology (URM) from Cadence, and Verification Methodology Manual (VMM) from Synopsys. Each one provided a base class library and TLM level modeling capabilities. Each one claimed to allow you to write reusable code. The end result was that the customer base was confused on which one to adopt. After intense debate on unifying all three base class libraries and methodologies, Accelera came forward and decided to pick OVM and build an industry standard methodology around it that was "universal." Most of the features of the language came from OVM, and thus UVM was born. All vendors now support this methodology and the standard class library that comes with the tool.



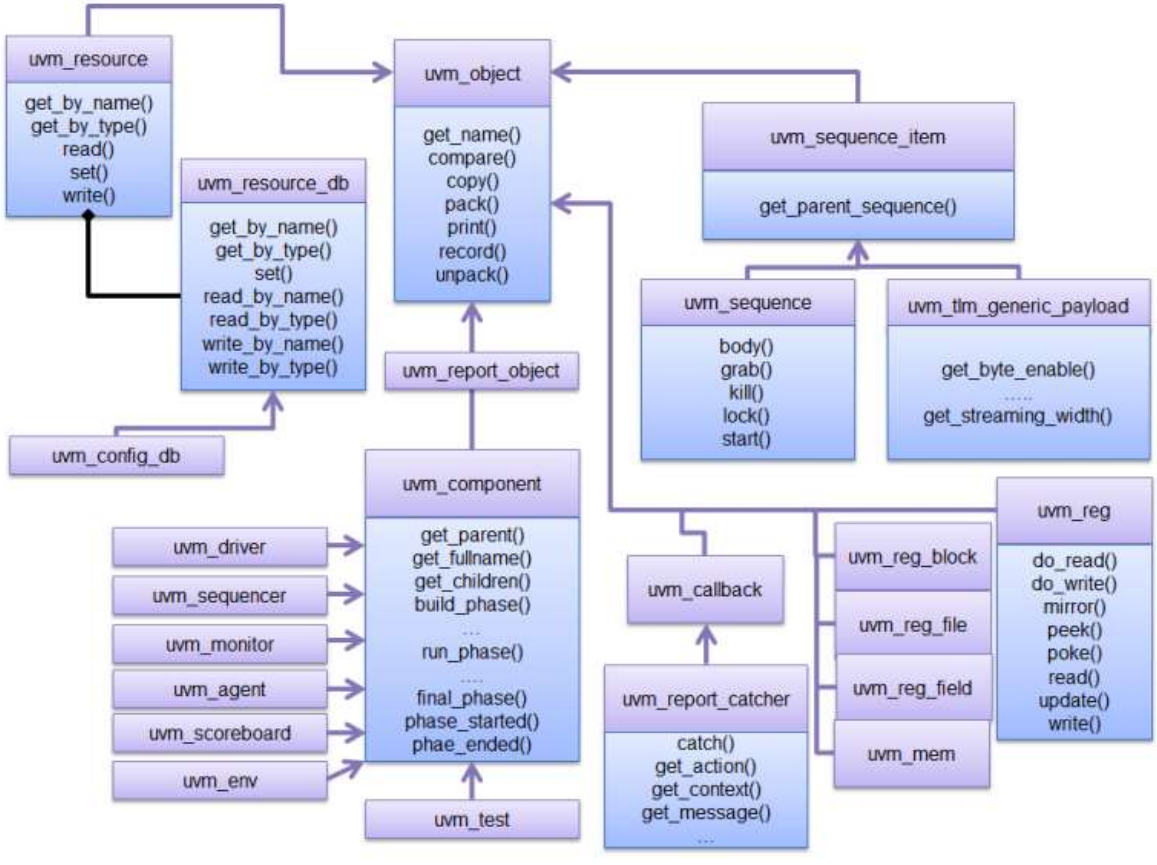
Why UVM?

With pure SystemVerilog, testbenches are written ad hoc since there is no coding standard there. These testbenches are then not quite reusable. The code is also hard to understand and maintain.

The primary advantage is that the methodology specifies and lays out a set of guidelines to be followed for creation of verification testbenches. This will ensure testbench uniformity between different verification teams, cross-compatibility between IP and standalone environment integration, flexibility and ease of maintaining testbenches.

2. UVM Class Hierarchy

The UVM Class Library provides all the building blocks you need to quickly develop, well-constructed, reusable, verification components and test environments. The library consists of base classes, utilities, and macros. The next figure shows a subset of those classes.



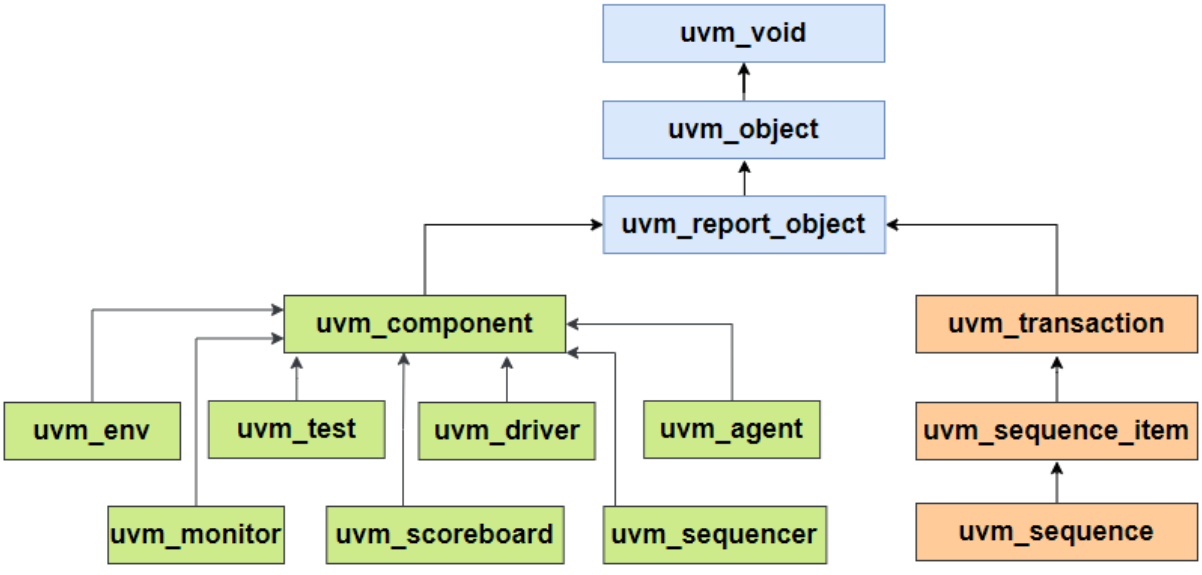
The advantages of using the UVM Class Library include:

- A robust set of built-in features: The UVM Class Library provides many features that are required for verification, including complete implementation of printing, copying, test phases, factory methods, and more.

- Correctly implemented UVM concepts: Each component in the UVM hierarchy can be derived from a corresponding UVM Class Library component. Using these base class elements increases the readability of your code since each component's role is predetermined by its parent class.

UVM classes have 3 major types:

- UVM object
- UVM transaction
- UVM component



They are two branches in the hierarchy:

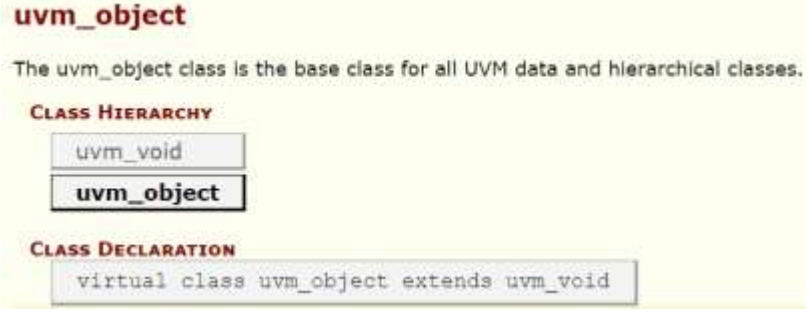
1. Component Branch: Classes that define verification components like driver, monitor and agents.
2. Sequence Branch: Classes that define data objects consumed and operated upon by verification components

uvm_void

The uvm_void class is the base class for all UVM classes. It is an abstract class with no data members or functions. It allows for generic containers of objects to be created.

uvm_object

The UVM object is a data structure used for testbench configuration and it is the base class available for component and sequence branch. The uvm_object provides methods like create, clone, copy, record, compare, print, etc.

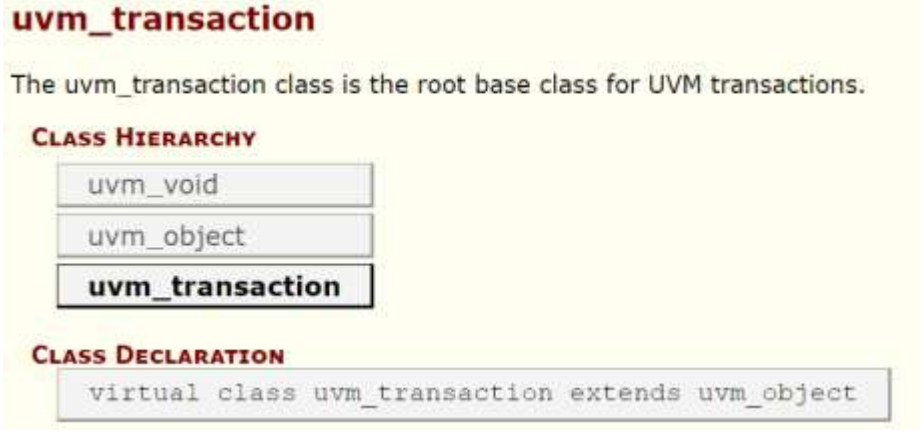


uvm_report_object

The uvm_report_object provides reporting functionality for UVM. The message, error, or warning prints are very important for debugging purposes that are being facilitated by the uvm_report_object class.

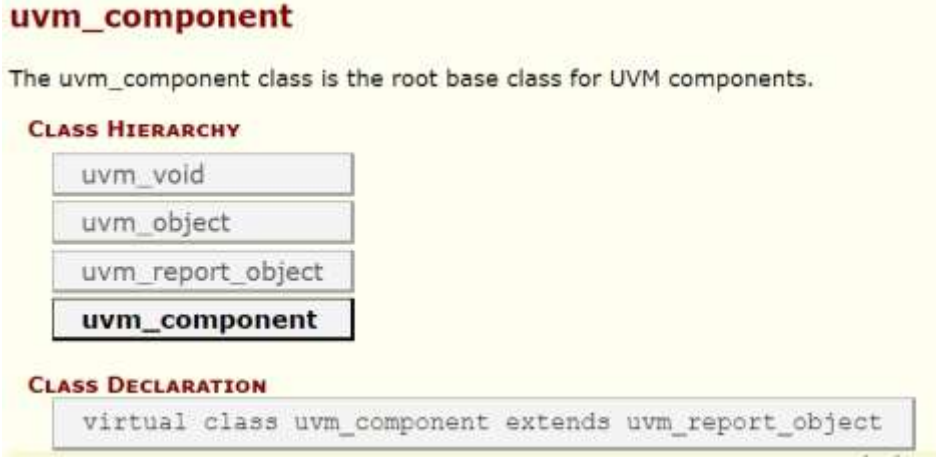
uvm_transaction

UVM transaction is used for generating stimulus and its analysis. They are transient in nature. The uvm_transaction class is inherited from uvm_object that adds additional information of a timing, notification events, and recording interface. The uvm_sequence_item class is derived from the uvm_transaction class that adds basic functionality for sequence and sequence items like get_sequence_id, set_sequencer, get_sequence, etc.



uvm_component

The uvm_component class is the root base class for all UVM components. Components are quasi-static objects that exist throughout simulation. For testbench hierarchy, base class components are available in UVM as uvm_env, uvm_agent, uvm_monitor, uvm_driver, uvm_sequencer, etc. Each component is uniquely addressable via a hierarchical path name.



Creates a new component with the given leaf instance name and handle to its parent. If the component is a top-level component, parent should be null and the component will become a child of the implicit top level component, uvm_top. All classes derived from uvm_component must call super.new(name,parent).

In addition to the features inherited from uvm_object and uvm_report_object, uvm_component provides Hierarchy, Phasing, Reporting, Transaction recording and Factory.

- **Hierarchy:** Provides methods for searching and traversing component hierarchy.
  Example:
  1. get_parent: It returns a handle for the current component's parent. If it has no parent, then it returns null.
  2. get_full_name: It returns a complete hierarchical name for this object.

- **Phasing:** UVM defines a set of simulation phases that enable users to control the order in which testbench components are created, initialized, and executed. This allows all components to execute in synchronization. **Will be discussed later.**
  Example: build_phase, connect_phase, run_phase, etc.

- **Reporting:** Provides an interface to uvm_report_handler to process all messages, errors, and warnings.
  Example: set_report_id_verbosity_hier, set_report_verbosity_level_hier

- **Objection:** Provides an interface to the uvm_objection mechanism.
  Example: raised, dropped, all_dropped, etc.

- **Configuration:** UVM provides a configuration database that allows users to store and retrieve configuration information for testbench components
  Example: print_config_settings, print_config, apply_config_settings, etc

- **Transaction recording:** Provides interface to record transactions consumed or produced by the component to a transaction database.
  Example: accept_tr, begin_tr, record_error_tr, tr_database, etc.

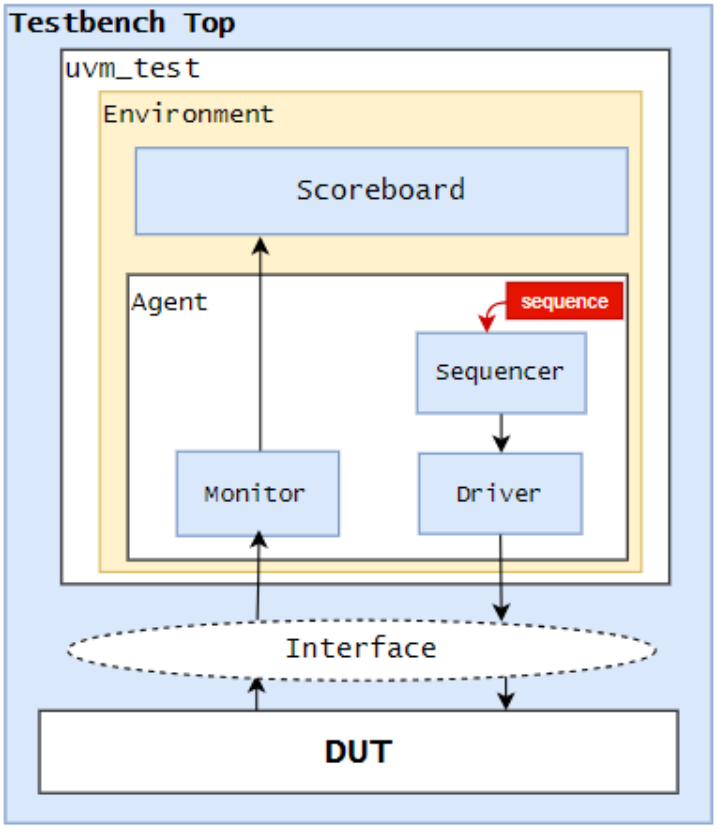
- **Factory:** Provides an interface to the uvm_factory to create new components and objects. This also allows an override mechanism for components and objects. **Will be discussed later.**
  Example: create_component, create_object, set_type_override_by_type, set_inst_override_by_type, etc

Difference between uvm_component and uvm_object:

- UVM components are non-transient but UVM objects are transients.
- UVM components are static in nature and exist throughout the simulation and UVM objects are dynamic in nature that has a limited lifetime in the simulation.
- `uvm_component_utils and `uvm_object_utils macros are used for factory registration for UVM components and objects respectively. They cannot be used interchangeably.
- Default constructor for uvm_component has two arguments: name and parent
- Default constructor for uvm_object has a single argument: name

3. UVM TestBench Hierarchy

The UVM employs a layered, object-oriented approach 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 DUT is connected to a layer of transactors (drivers, monitors). 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 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, sequencers, etc. All structural elements in a UVM testbench are extended from the uvm_component base class.

The Agent(s) and other design specific components are encapsulated in a uvm_env which is in turn instantiated and customize by a top-level uvm_test component.

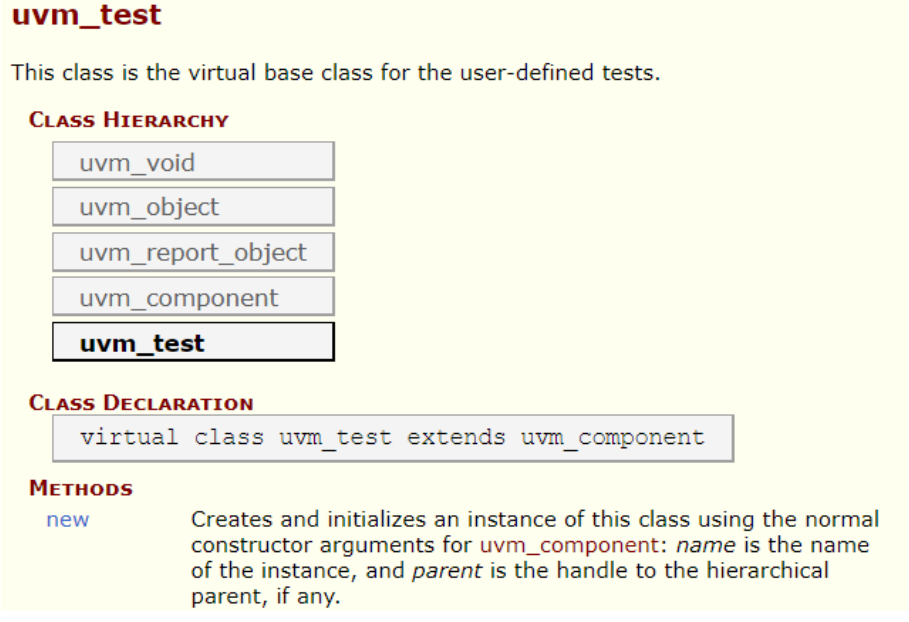
3.1 UVM Testbench Top

The testbench top is a static container that has an instantiation of DUT and interfaces. The interface instance connects with DUT signals in the testbench top. The clock is generated and initially reset is applied to the DUT. It is also passed to the interface handle. An interface is stored in the uvm_config_db using the set method and it can be retrieved down the hierarchy using the get method. UVM testbench top is also used to trigger a test using run_test() call.

```
`include "uvm_macros.svh"
import uvm_pkg::*;

module tb_top;
  bit clk;
  bit reset;
  always #5 clk = ~clk;

  initial begin
    clk = 0;
    reset = 1;
    #5;
    reset = 0;
  end
  add_if vif(clk, reset);

  // Instantiate design top
  adder DUT(.clk(vif.clk),
            .reset(vif.reset),
            .in1(vif.ip1),
            .in2(vif.ip2),
            .out(vif.out)
           );

  initial begin
    // set interface in config_db
    uvm_config_db#(virtual add_if)::set(uvm_root::get(), "*",
"vif", vif);
    // Dump waves
    $dumpfile("dump.vcd");
    $dumpvars;
  end
  initial begin
    run_test("base_test");
  end
endmodule
```

3.2 UVM Test

The UVM Test is the top-level UVM Component in the UVM Test bench. The UVM Test Instantiates the top-level environment, configures the environment (via factory overrides or the configuration database), and applies stimulus by invoking UVM Sequences through the environment to the DUT (start sequences over sequencers). There is a one base UVM Test with the UVM Environment instantiation and other common items. Then, other individual tests will extend this base test and configure the environment differently or select different sequences to run. The user-defined test class is derived from uvm_test.



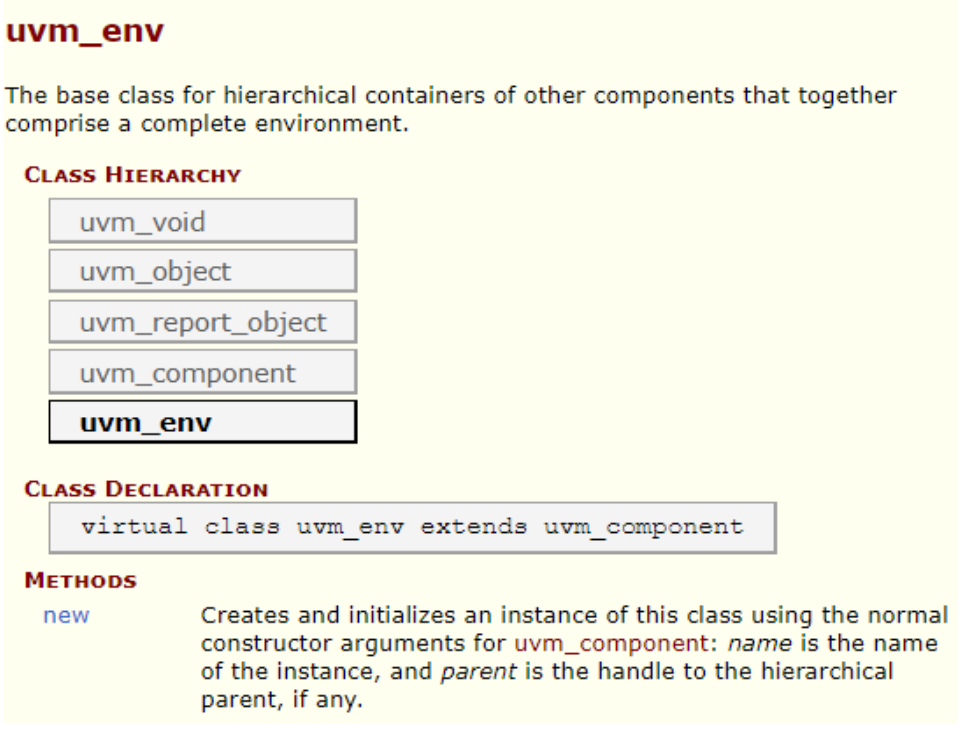
Steps to write a test case

1. Create a user-defined test class extended from uvm_test and register it in the factory.
2. Declare environment, sequence handle, and configuration objects based on the requirement.
3. Write standard new() function. Since the test is a uvm_component. The new() function has two arguments as string name and uvm_component parent.
4. Implement build_phase to create instances of environment, sequence classes, and set configurable objects in the configurable database.
5. Implement run_phase to start sequences on required sequencers with raise/drop objection callbacks. The raise_objection is called prior to executing the run_phase. Once activities of run_phase are completed, the drop_objection is called. The run_phase of the test is mentioned below.

```
class my_test extends uvm_test;
  env env_o;
  base_seq bseq;
  `uvm_component_utils(my_test)

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

  function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    env_o = env::type_id::create("env_o", this);
  endfunction

  task run_phase(uvm_phase phase);
    phase.raise_objection(this);
    bseq = base_seq::type_id::create("bseq");

    repeat(10) begin
      #5; bseq.start(env_o.agt.seqr);
    end

    phase.drop_objection(this);
    `uvm_info(get_type_name, "End of testcase", UVM_LOW);
  endtask
endclass
```

3.3 UVM Environment

The UVM Environment is a higher-level verification and hierarchical component that groups together other verification components that are interrelated. It consists of many components like UVM Agents, UVM Scoreboards, or even other UVM Environments. The top-level UVM Environment encapsulates all the verification components targeting the DUT. The configuration of the environment enables customization of its topology and behavior according to the specific UVM test. The function of a single environment is the generation of the constrained random traffic to stimulate the DUT, monitoring of the DUT response, check of ongoing traffic with respect to protocol specifications and the coverage collection. The user-defined env class has to be extended from the uvm_env class.



How to create a UVM env?:

1. Create a user-defined env class extended from uvm_env and register it in the factory.
2. In the build_phase, instantiate the agent, other verification components, and use the configuration database to set/get configuration variables.
3. In the connect_phase, connect the monitor, scoreboard, functional coverage and other subscribers using the TLM interface.

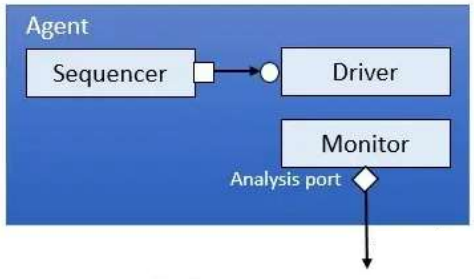
```
class env extends uvm_env;
  `uvm_component_utils(env)
  agent agt;
  scoreboard sb;
  func_cov fcov;

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

  function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    agt = agent::type_id::create("agt", this);
    sb = scoreboard::type_id::create("sb", this);
    fcov = func_cov::type_id::create("fcov", this);
  endfunction

  function void connect_phase(uvm_phase phase);
    // connect agent and scoreboard using TLM interface
    // Ex. agt.mon.item_collect_port.connect(sb.item_collect_export);
  endfunction
endclass
```

3.4 UVM Agent

The UVM agent is a hierarchical component that contains other verification components that are dealing with a specific DUT interface. A typical UVM Agent includes a UVM Sequencer, a UVM Driver, and a UVM Monitor. UVM Agents might include other components, like coverage collectors, protocol checkers, a TLM model, etc. Although the sequencers, drivers and monitors can be reused independently, in order to improve the interoperability, a single agent can stimulate and verify the DUT. However, the verification components may contain a few agents. The agent can initiate the transactions to the DUT or react to the transaction requests. The UVM distinguishes either active or passive agents.

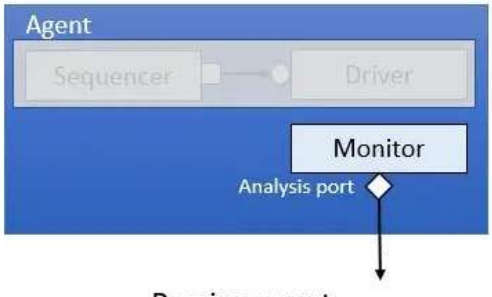
There are two types of agents

- Active agent: An Active agent stimulates the DUT by driving transactions according to the test scenario and monitors the device. It instantiates all three components driver, monitor, and sequencer.



Active agent

- Passive agent :A passive agent does not drive stimulus to the DUT. It instantiates only a monitor component. It is used as a sample interface for coverage and checker purposes.



Passive agent

How to create a UVM agent?

1. Create a user-defined agent class extended from uvm_agent and register it in the factory.
2. In the build_phase, instantiate driver, monitor, and sequencer if it is an active agent. Instantiate monitor alone if it is a passive agent.
3. In the connect_phase, connect driver and sequencer components.

The get_is_active() function is used to find out the type of agent. The driver, sequencer instances are created if it is an active agent and monitor instance can be created by default irrespective of agent type.

The get_is_active() function returns an enum as UVM_ACTIVE or UVM_PASSIVE of type uvm_active_passive_enum.

```
class a_agent extends uvm_agent;
  driver drv;
  sequencer seqr;
  monitor_A mon_A;
  `uvm_component_utils(a_agent)

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

  function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    if(get_is_active() == UVM_ACTIVE) begin
      drv = driver::type_id::create("drv", this);
      seqr = sequencer::type_id::create("seqr", this);
      `uvm_info(get_name(), "This is Active agent", UVM_LOW);
    end
    mon_A = monitor_A::type_id::create("mon_A", this);
  endfunction

  function void connect_phase(uvm_phase phase);
    super.connect_phase(phase);
    if(get_is_active() == UVM_ACTIVE)
      drv.seq_item_port.connect(seqr.seq_item_export);
  endfunction
endclass
```

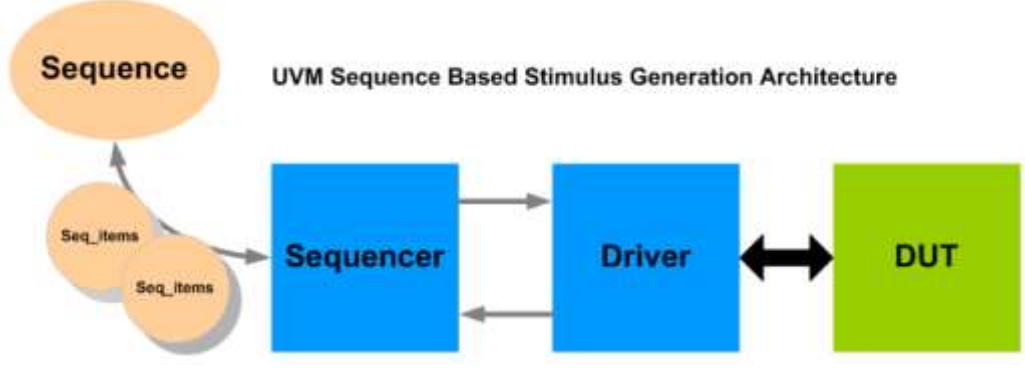
3.5 UVM Sequence Items

All user-defined sequence items are extended from the uvm_sequence_item class as it leverages generating stimulus and has control capabilities for the sequence-sequencer mechanism.

```
class seq_item extends uvm_seqeunce_item;
  rand int        value;
  rand color_type colors;
  rand byte       data[4];
  rand bit [7:0]  addr;
  `uvm_object_utils_begin(my_object)
  `uvm_field_int(value, UVM_ALL_ON)
  `uvm_field_string(names, UVM_ALL_ON)
  `uvm_field_enum(color_type, colors, UVM_ALL_ON)
  `uvm_field_sarray_int(data, UVM_ALL_ON)
  `uvm_field_int(addr, UVM_ALL_ON)
  `uvm_object_utils_end

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

3.6 UVM Sequence

UVM sequence is a container that holds data items (uvm_sequence_items) which are sent to the driver via the sequencer. UVM Sequences are not part of the component hierarchy. UVM Sequences can be transient or persistent. A UVM Sequence instance can come into existence for a single transaction, it may drive stimulus for the duration of the simulation, or anywhere in between. UVM Sequences can operate hierarchically with one sequence, called a parent sequence, invoking another sequence, called a child sequence. To operate, each UVM Sequence is eventually bound to a UVM Sequencer. Multiple UVM Sequence instances can be bound to the same UVM Sequencer.



How to write a sequence?

An intention is to create a seq_item, randomize it, and then send it to the driver. To perform this operation any one of the following approaches is followed in the sequence:

1. Using macros like `uvm_do , `uvm_create, `uvm_send etc
2. Using existing methods from the base class

   a. Using wait_for_grant(), send_request(), wait_for_item_done()

   b. Using start_item/finish_item methods.

```
class my_sequence extends uvm_sequence #(my_seq_item);
  `uvm_object_utils(my_sequence)

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

  task body();
    ...
  endtask
endclass
```

```
// Using `uvm_do
task body();
  `uvm_do(seq1); // calling seq1
  `uvm_do(seq2); // calling seq2
Endtask
```

```
// Using `uvm_create, `uvm_send etc
task body();
  `uvm_create(req);
   assert(req.randomize());
  `uvm_send(req);
endtask
```

```
// Using existing methods from the base class:
      wait_for_grant(), send_request(),
      wait_for_item_done()

task body();
     req = seq_item::type_id::create("req");
     wait_for_grant();
     assert(req.randomize());
     send_request(req);
     wait_for_item_done();
     get_respose(rsp);
endtask
```

Steps:

1. Create a seq_tem using create() method.
2. wait_for_grant
3. Randomize seq_item
4. send_request(req)
5. wait_for_item_done()
6. get_response (rsp)

```
// Using existing methods from the base class:
      start_item/finish_item
task body();
     req = seq_item::type_id::create("req");
     start_item(req);
     assert(req.randomize());
     finish_item(req);
endtask;
```

Steps:

1. Create a seq_tem using create() method.
2. start_item(req)
3. Randomize seq_item
4. finish_item(req)

3.7 UVM Sequencer

The UVM sequencer behaves as an arbiter for controlling transaction flow from multiple stimulus sequences. More specifically, the UVM Sequencer controls the flow of UVM Sequence Items transactions generated by one or more UVM Sequences.

A user-defined sequencer is recommended to extend from the parameterized base class "uvm_sequencer" which is parameterized by request (REQ) and response (RSP) item types. Response item usage is optional. So, mostly sequencer class is extended from a base class that has only a REQ item.

```
class my_sequencer extends uvm_sequencer #(data_item);
  `uvm_component_utils(my_sequencer)

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

uvm_sequencer #(REQ,RSP)

CLASS HIERARCHY

- uvm_void
- uvm_object
- uvm_report_object
- uvm_component
- uvm_sequencer_base
- uvm_sequencer_param_base#(REQ,RSP)
- **uvm_sequencer#(REQ,RSP)**

CLASS DECLARATION

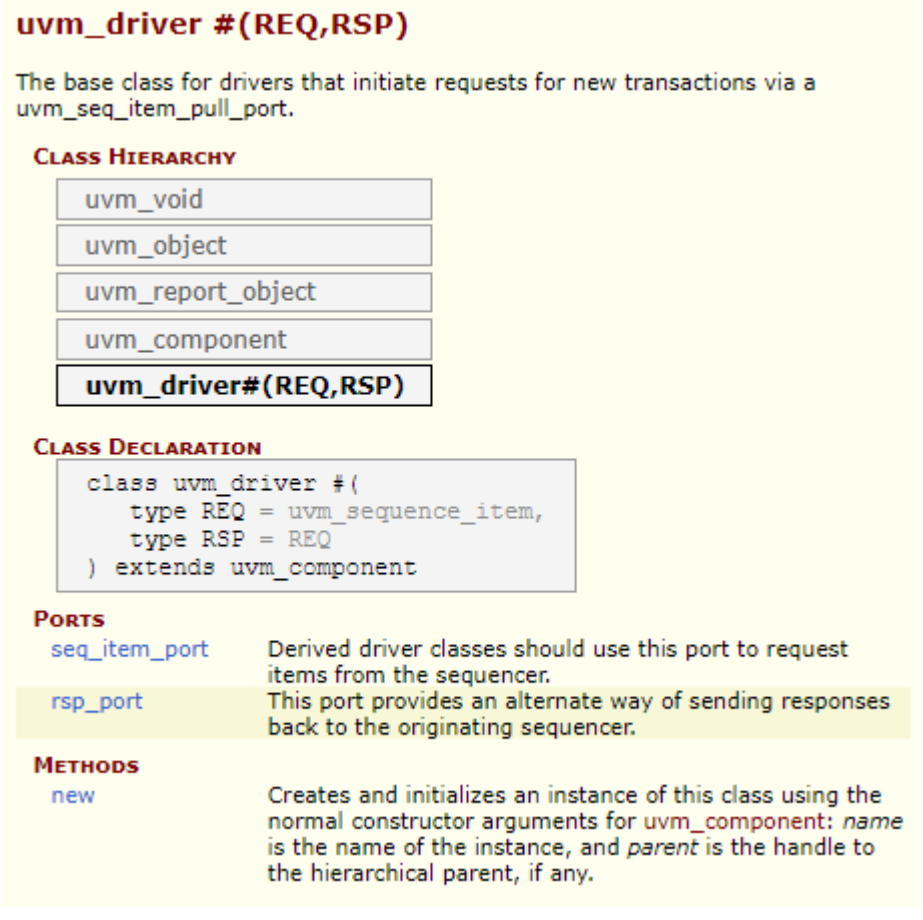
```
class uvm_sequencer #(
    type REQ = uvm_sequence_item,
         RSP = REQ
) extends uvm_sequencer_param_base #(REQ, RSP)
```

new	Standard component constructor that creates an instance of this class using the given *name* and *parent*, if any.
stop_sequences	Tells the sequencer to kill all sequences and child sequences currently operating on the sequencer, and remove all requests, locks and responses that are currently queued.
SEQUENCER INTERFACE	This is an interface for communicating with sequencers.
seq_item_export	This export provides access to this sequencer's implementation of the sequencer interface.
get_next_item	Retrieves the next available item from a sequence.
try_next_item	Retrieves the next available item from a sequence if one is available.
item_done	Indicates that the request is completed.
put	Sends a response back to the sequence that issued the request.
get	Retrieves the next available item from a sequence.
peek	Returns the current request item if one is in the FIFO.
wait_for_sequences	Waits for a sequence to have a new item available.
has_do_available	Returns 1 if any sequence running on this sequencer is ready to supply a transaction, 0 otherwise.

3.8 UVM Driver

The UVM Driver receives the sequence Item transactions from the UVM Sequencer and applies it on the DUT Interface. Thus, a UVM Driver extends abstraction levels by converting transaction-level stimulus into pin-level stimulus. The driver has to be extended from uvm_component names as uvm_driver. The transaction or sequence items are retrieved from the sequencer and the driver drives them to the design using the interface handle. An interface handle can be retrieved from the configuration database which was already set in the top-level hierarchy.

The uvm_driver class is a parameterized class of type REQ sequence_item and RSP sequence item. RSP sequence item is optional.



uvm_driver #(REQ,RSP)

The base class for drivers that initiate requests for new transactions via a uvm_seq_item_pull_port.

CLASS HIERARCHY

- uvm_void
- uvm_object
- uvm_report_object
- uvm_component
- **uvm_driver#(REQ,RSP)**

CLASS DECLARATION

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

PORTS

seq_item_port	Derived driver classes should use this port to request items from the sequencer.
rsp_port	This port provides an alternate way of sending responses back to the originating sequencer.

METHODS

new	Creates and initializes an instance of this class using the normal constructor arguments for uvm_component: *name* is the name of the instance, and *parent* is the handle to the hierarchical parent, if any.

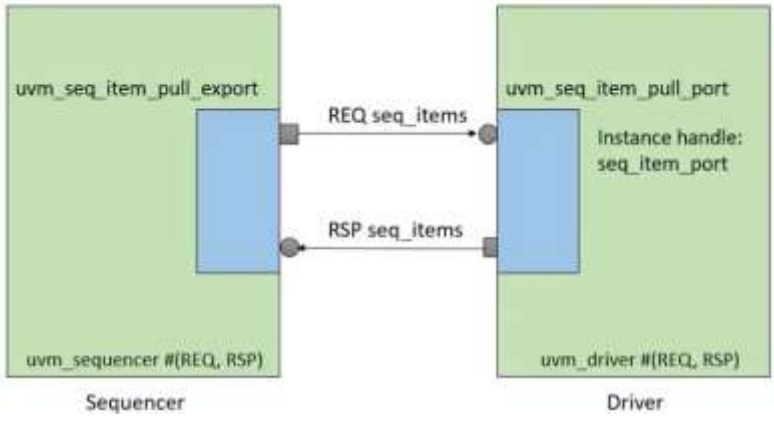
How to write driver code?

1. Create a user-defined driver class extended from uvm_driver and register it in the factory.
2. Declare virtual interface handle to retrieve actual interface handle using configuration database in the build_phase.
3. Write standard new() function. Since the driver is a uvm_component. The new() function has two arguments as string name and uvm_component parent.
4. Implement build_phase and get interface handle from the configuration database.
5. Implement run_phase to get the sequence items and drive them to the DUT using a virtual interface handle.

```
class driver extends uvm_driver#(seq_item);
  virtual add_if vif;
  `uvm_component_utils(driver)

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

  function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    if(!uvm_config_db#(virtual add_if) :: get(this, "", "vif", vif))
      `uvm_fatal(get_type_name(), "Not set at top level");
  endfunction

  task run_phase (uvm_phase phase);
    // Get the sequence_item and drive it to DUT
  endtask
endclass
```

The sequencer and driver communicate with each other using a bidirectional TLM interface to transfer REQ and RSP sequence items.

The driver has uvm_seq_item_pull_port which is connected with uvm_seq_item_pull_export of the associated sequencer. Both uvm_seq_item_pull_port and uvm_seq_item_pull_export are parameterized classes with REQ and RSP sequence items. The TLM connection between driver and sequencer is **one-to-one connection**. It means neither multiple sequencers are connected to a single driver nor multiple drivers connected to a single sequencer.



There are multiple mechanisms that allow the driver to get a series sequence items from sequencer and respond back after data has been driven into DUT.

Connection Methods:

1. Using get_next_item/ try_next_item and item_done
2. Using get and put

```
// Using get_next_item, try_next_item and item_done methods
task run_phase (uvm_phase phase);
  forever begin
    seq_item_port.get_next_item(req);
    // Driving logic
    ...
    seq_item_port.item_done();
  end
endtask
```

```
// Using get and put methods.
task run_phase (uvm_phase phase);
  forever begin
    seq_item_port.get(req);
    // Driving logic
    ...
    seq_item_port.put(rsp_item);
  end
endtask
```

Difference between get_next_item/ item_done and get/ put approach?

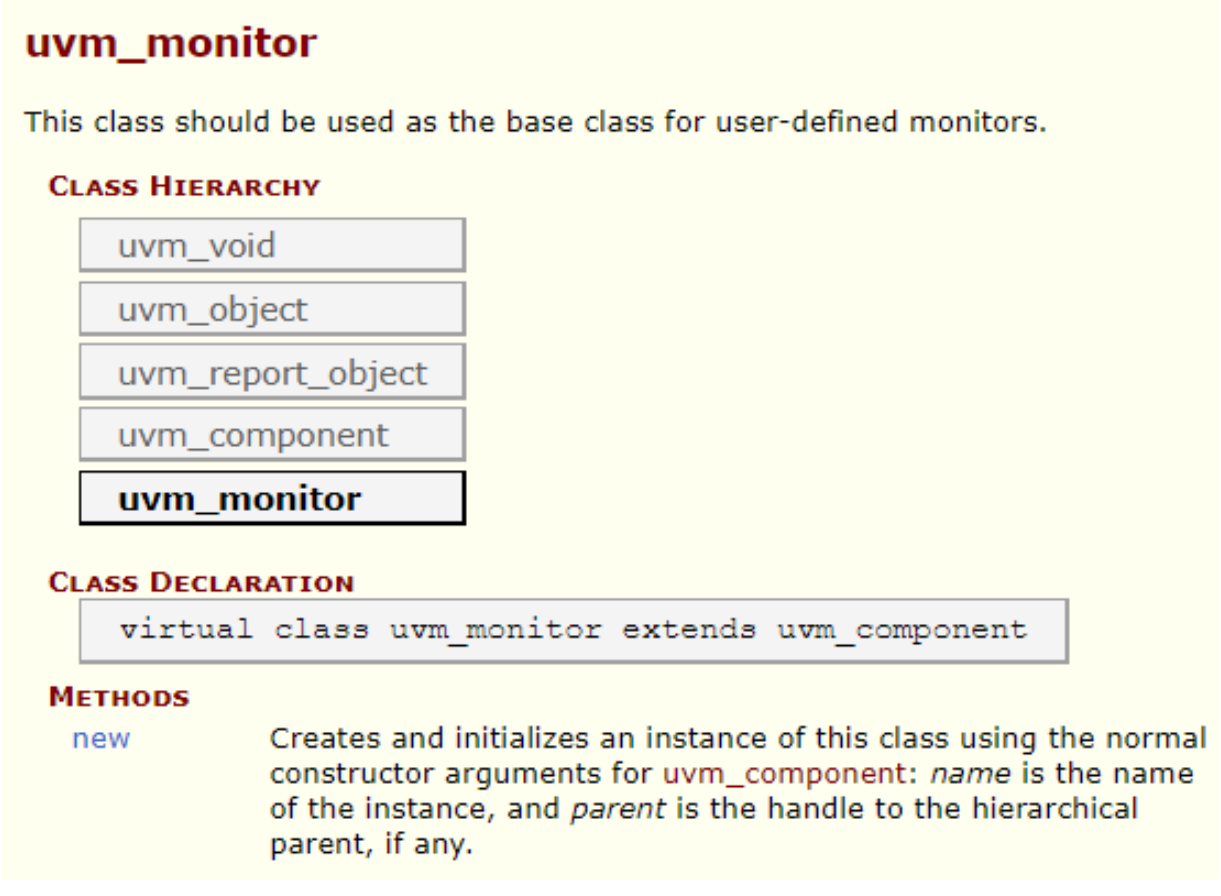
1. The item_done must be called after get_next_item() or successful try_next_item() call, then only the next sequence item can be requested. Whereas, get() call can request another request item even if the put() method is not called.
2. The put() call must be called with the RSP sequence item as an argument whereas it is optional for item_done() call.
3. In the case of get_next_item/ item_done approach, wait_for_item_done task (uvm_sequence_base class method) gets unblocked when item_done is called from the driver once transactions are driven to the DUT using a virtual interface. In the case of get/put approach, the wait_for_item_done gets unblocked when a put method is called from the driver where a driver has had any time to process and drive the sequence item.
4. Since the sequence writer has to remember the handle of the response item, the get/put approach is more complicated to implement than the get_next_item/ item_done approach.

3.9 UVM Monitor

A UVM monitor is a passive component used to capture DUT signals using a virtual interface and translate them into a sequence item format. These sequence items or transactions are broadcasted to other components like the UVM scoreboard, coverage collector, etc. It uses a TLM analysis port to broadcast transactions.

As we said before uvm_seq_item_pull_port and uvm_seq_item_pull_export are one to one communication and in case of monitor that broadcasted sequences to many components, we cannot do it using the old way. Instead we use uvm_analysis_port. The idea behind having an analysis port is that a component like monitor should be able to generate a stream of transactions regardless of whether there is a target actually connected to it.

The uvm_analysis_port is a specialized TLM based class whose interface consists of a single function write () and can be embedded within any component. This port contains a list of analysis exports that are connected to it, When the component (my monitor) calls analysis_port.write() , it basically cycles through the list and calls the write() method of each connected export. If nothing is connected to it, then it simply does not do anything. Also it's worth to note that write() is a void function and hence will always complete within the same simulation delta cycle.



uvm_monitor

This class should be used as the base class for user-defined monitors.

CLASS HIERARCHY

uvm_void
uvm_object
uvm_report_object
uvm_component
uvm_monitor

CLASS DECLARATION

```
virtual class uvm_monitor extends uvm_component
```

METHODS

new	Creates and initializes an instance of this class using the normal constructor arguments for uvm_component: *name* is the name of the instance, and *parent* is the handle to the hierarchical parent, if any.

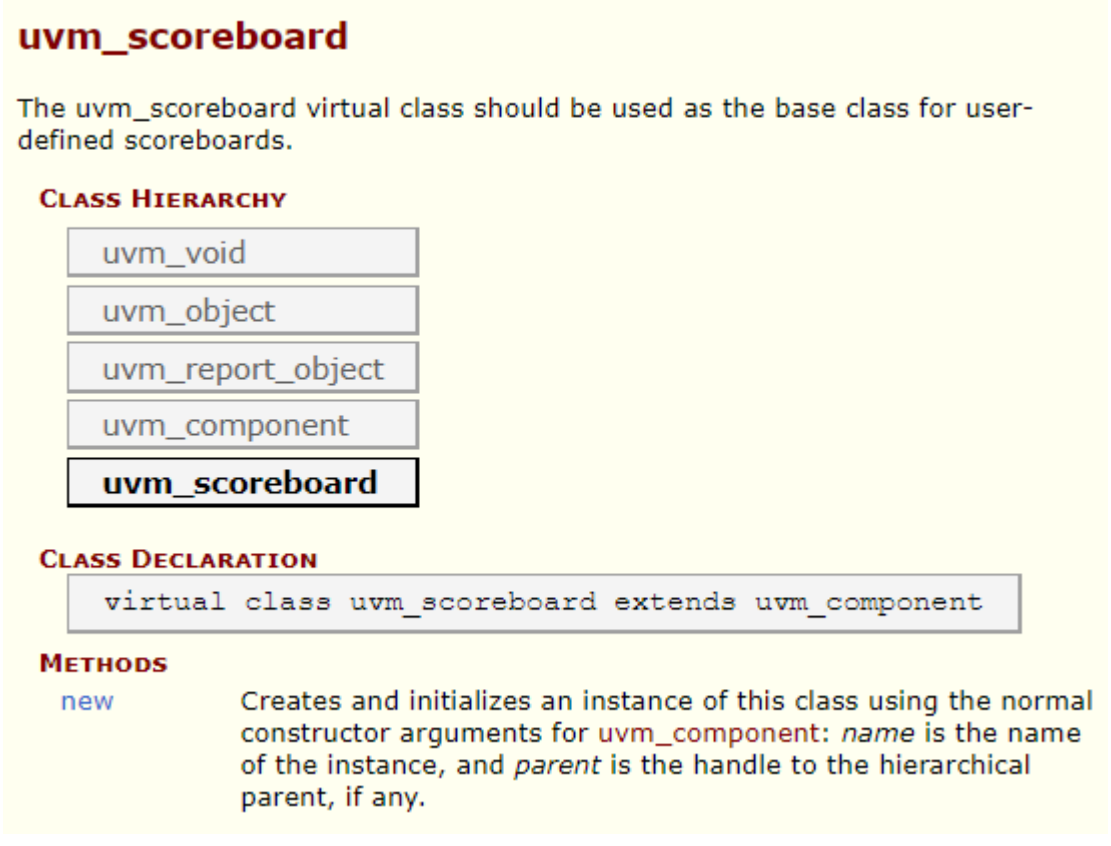
How to create a UVM monitor?

1. Create a user-defined monitor class extended from uvm_monitor and register it in the factory.
2. Declare virtual interface handle to retrieve actual interface handle using configuration database in the build_phase.
3. Declare analysis port to broadcast the sequence items or transactions.
4. Write standard new() function. Since the monitor is a uvm_component. The new() function has two arguments as string name and uvm_component parent.
5. Implement build_phase and get interface handle from the configuration database.
6. Implement run_phase to sample DUT interface using a virtual interface handle and translate into transactions. The write() method sends transactions to the collector component.

```
class monitor extends uvm_monitor;
  // declaration for the virtual interface, analysis port, and
monitor sequence item.
  virtual add_if vif;
  uvm_analysis_port #(seq_item) item_collect_port;
  seq_item mon_item;
  `uvm_component_utils(monitor)

  // constructor
  function new(string name = "monitor", uvm_component parent = null);
    super.new(name, parent);
    item_collect_port = new("item_collect_port", this);
    mon_item = new();
  endfunction

  function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    if(!uvm_config_db#(virtual add_if) :: get(this, "", "vif", vif))
      `uvm_fatal(get_type_name(), "Not set at top level");
  endfunction

  task run_phase (uvm_phase phase);
    forever begin
      // Sample DUT information and translate into transaction
      item_collect_port.write(mon_item);
    end
  endtask
endclass
```

3.10 UVM Scoreboard

The UVM Scoreboard checks the behavior of a certain DUT. It usually receives transactions carrying inputs and outputs of the DUT through UVM Agent analysis ports, runs the input transactions through a reference model (also known as the predictor) to produce expected transactions, and then compares the expected output versus the actual output.



uvm_scoreboard

The uvm_scoreboard virtual class should be used as the base class for user-defined scoreboards.

CLASS HIERARCHY

- uvm_void
- uvm_object
- uvm_report_object
- uvm_component
- **uvm_scoreboard**

CLASS DECLARATION

```
virtual class uvm_scoreboard extends uvm_component
```

METHODS

new	Creates and initializes an instance of this class using the normal constructor arguments for uvm_component: *name* is the name of the instance, and *parent* is the handle to the hierarchical parent, if any.

How to write scoreboard code in UVM?

1. Create a user-defined scoreboard class extended from uvm_scoreboard and register it in the factory.
2. Declare an analysis export to receive the sequence items or transactions from the monitor.
3. Write standard new() function. Since the scoreboard is a uvm_component. The new() function has two arguments as string name and uvm_component parent.
4. Implement build_phase and create a TLM analysis export instance.
5. Implement a write method to receive the transactions from the monitor.
6. Implement run_phase to check DUT functionality throughout simulation time.

```
class scoreboard extends uvm_scoreboard;
  uvm_analysis_imp #(seq_item, scoreboard) item_collect_export;
  seq_item item_q[$];
  `uvm_component_utils(scoreboard)

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

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

  function void write(seq_item req);
    `uvm_info(get_type_name, $sformatf("Received transaction =
%s", req), UVM_LOW);
    item_q.push_back(req);
  endfunction

  task run_phase (uvm_phase phase);
    seq_item sb_item;
    forever begin
      wait(item_q.size > 0);

      if(item_q.size > 0) begin
        sb_item = item_q.pop_front();
        // Checking comparing logic
        ...
      end
    end
  endtask
endclass
```

References:

- https://verificationacademy.com/verification-methodology-reference/uvm/docs_1.2/html/index.html
- https://www.chipverify.com/tutorials/uvm
- https://vlsiverify.com/uvm/uvm-class-hierarchy/
- https://colorlesscube.com/uvm-guide-for-beginners/