The American University in Cairo

CENTER OF NANOELECTRONICS & DEVICES

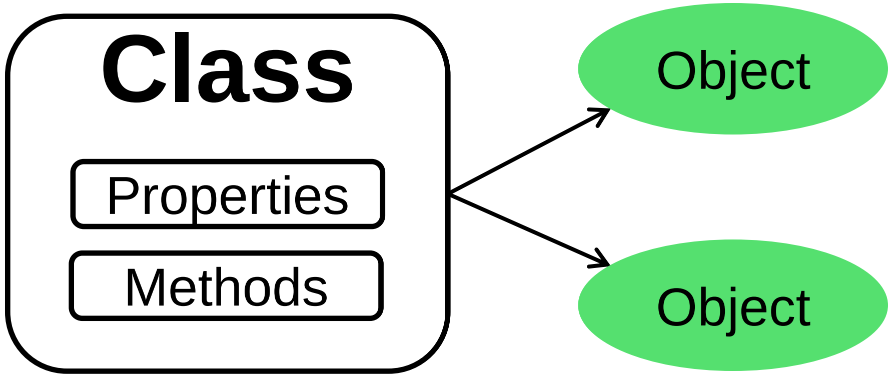
CND212: Digital Testing and Verification

OOP Revision

- Class encapsulates:
  - Variables (properties)
  - Subroutines (methods)

  } **Class members**

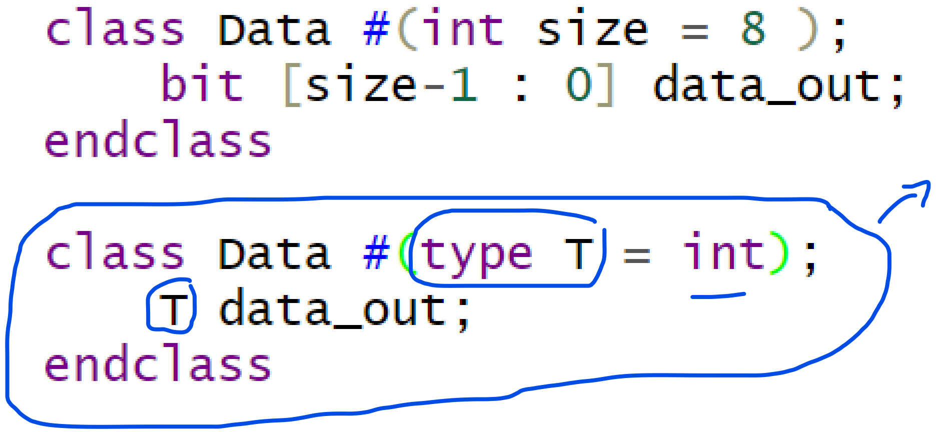
- **Object**: instance of a class.
- Class constructors: Allocate memory for the object using the **new()** method.
- Object members are accessed using the (.) notation.



OOP: Parameterized Classes

- Parameters used for objects customization during compile time.

- Can be value parameter or type parameter.



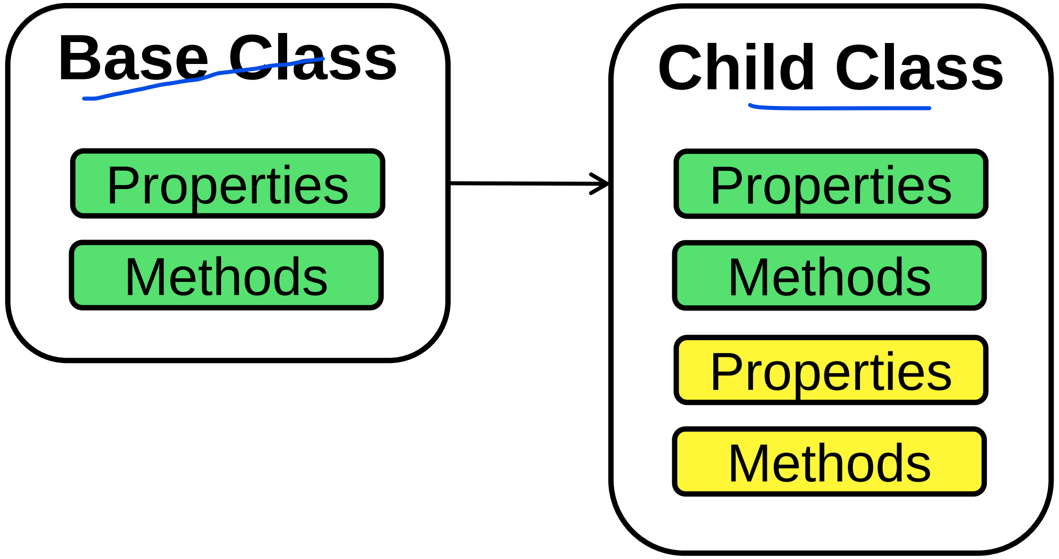
```
class Data #(int size = 8 );
    bit [size-1 : 0] data_out;
endclass

class Data #(type T = int);
    T data_out;
endclass
```

can be changed to float

OOP: Inheritance

- Any class can be extended to add more properties and/or methods

- If the method of the parent class is overridden in the child class, then using the 'super' keyword parent class method can be accessed from the child class.



OOP: Remember

- **Static variables** has one copy in all class instances.

- **Static methods** can be called without class instantiation and has access to static members only.

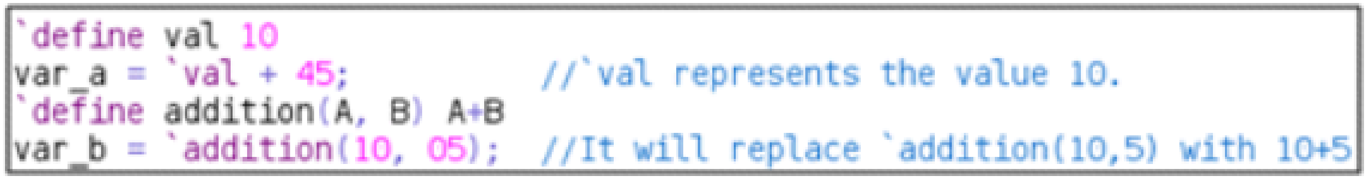
- **Virtual class** is a class that you cannot create an object from.

- **Virtual methods**: in case of identical function names between base and child class, using virtual methods allow access to the child class methods.

- **this keyword**: predefined object handle that refers to the current object

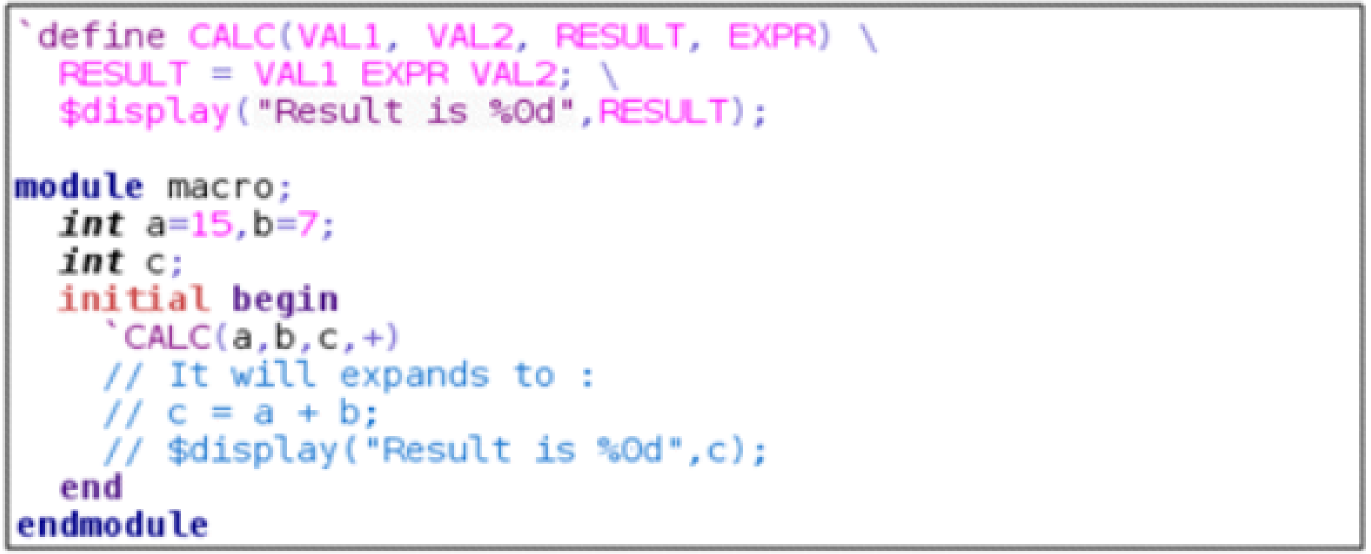
OOP: `define macro

- `define macro in systemverilog is a directive that allows you to define a shorthand or alias for a block of code.



```
`define val 10
var_a = `val + 45;          //`val represents the value 10.
`define addition(A, B) A+B
var_b = `addition(10, 05);  //It will replace `addition(10,5) with 10+5
```

Single line macro



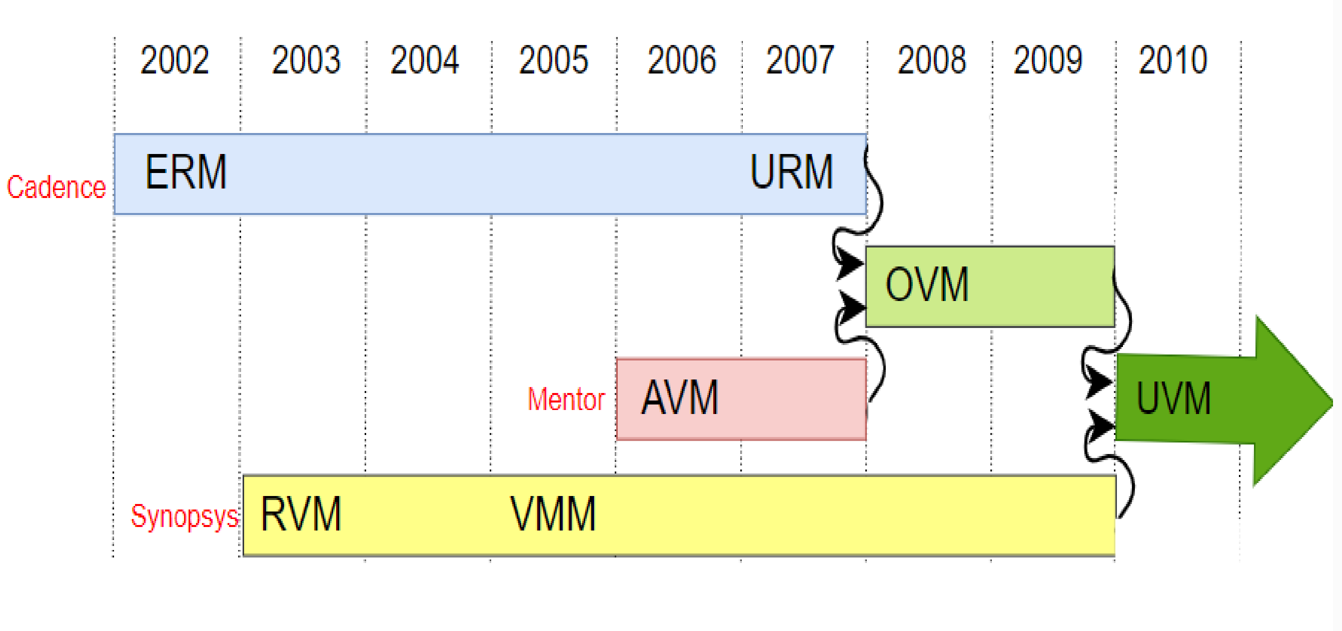
```
`define CALC(VAL1, VAL2, RESULT, EXPR) \
  RESULT = VAL1 EXPR VAL2; \
  $display("Result is %0d",RESULT);

module macro;
  int a=15,b=7;
  int c;
  initial begin
    `CALC(a,b,c,+)
    // It will expands to :
    // c = a + b;
    // $display("Result is %0d",c);
  end
endmodule
```

Multiline macro:

Introduction to UVM

UVM: What is UVM?

- Universal Verification Methodology.
- Class base library defined using systemverilog.
- Maintained by Accellera.
- Supported by multiple EDA vendors (Mentor, Synopsys, Cadence)

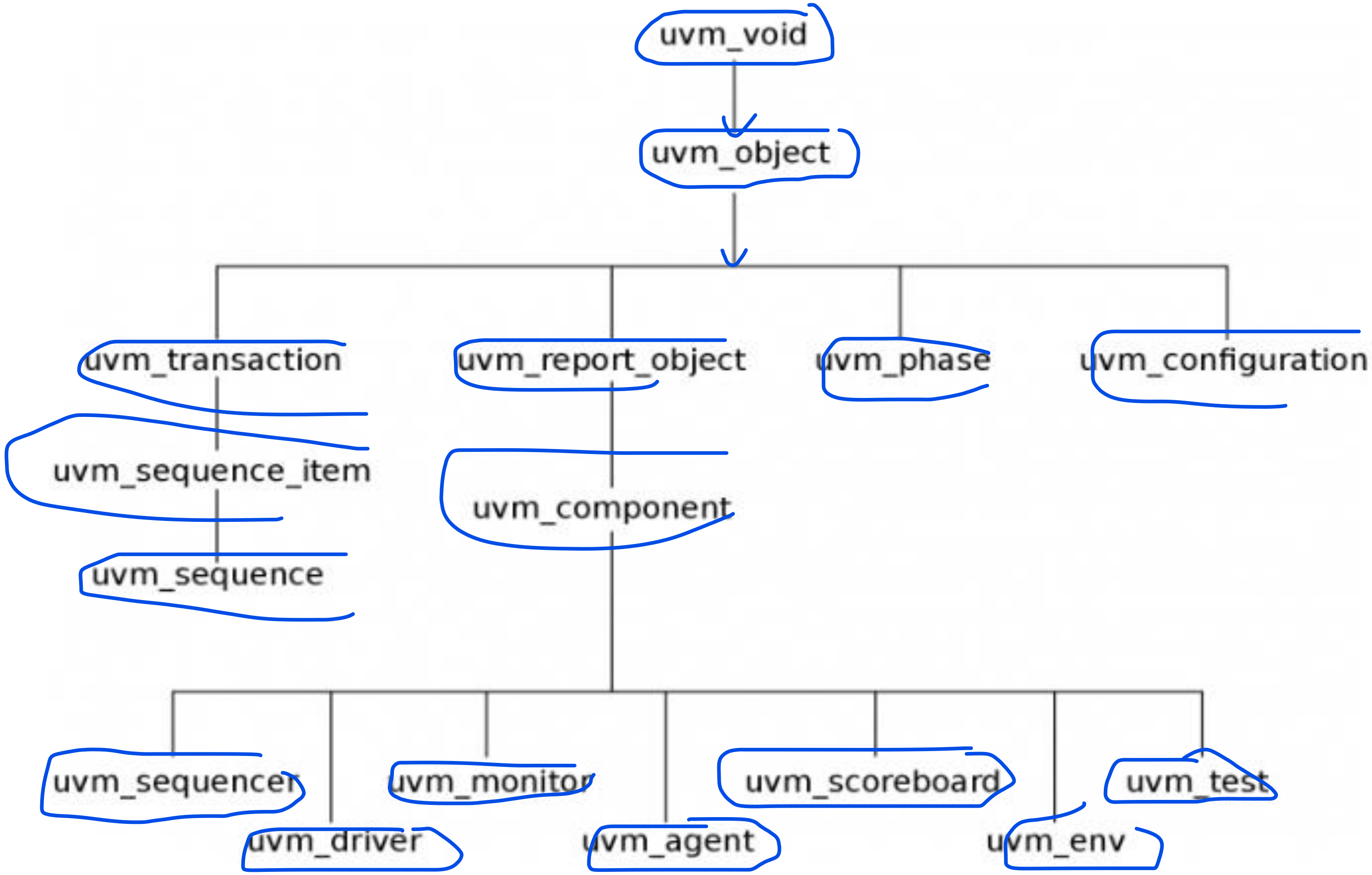


Why UVM?

- Pre-defined base classes library with built in methods
- Standardized methodology
- Reusability
- Modularity
- Scalability
- Configurability
- Separate test from testbench

UVM Class Hierarchy

- The UVM Class Library provides all the building blocks needed to quickly developed, well-constructed, reusable, testbench components.



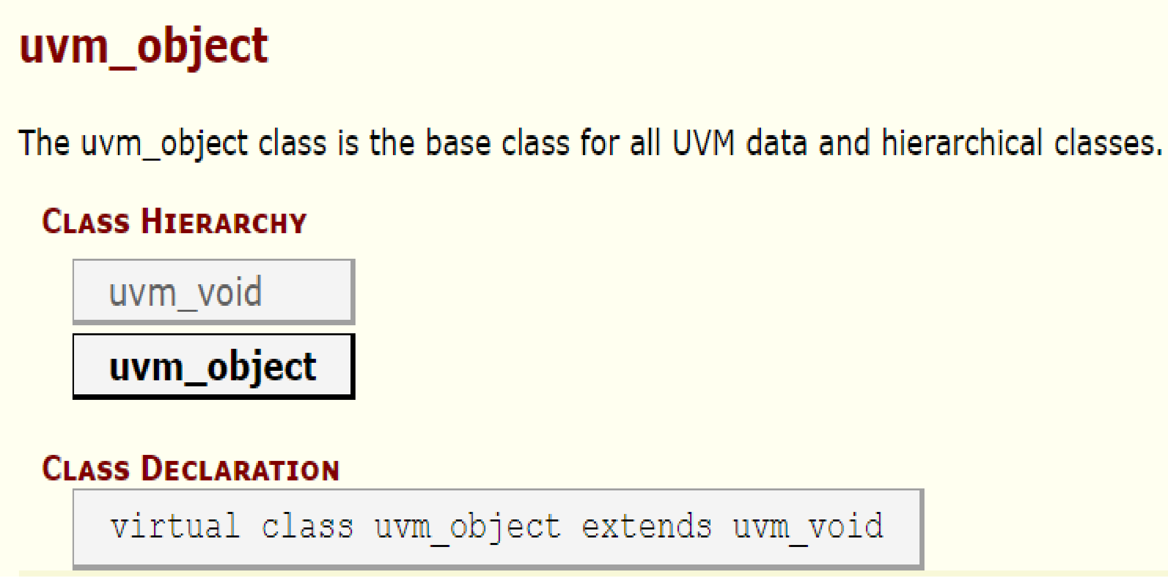
UVM Class Hierarchy

UVM void

- The base class for all UVM classes.
- It is an abstract class with no data members or functions.

UVM object

- The base class available for component and sequence branch.
- Provides methods like create, clone, copy, record, compare, print, etc.



uvm_object

The uvm_object class is the base class for all UVM data and hierarchical classes.

Class Hierarchy

uvm_void

uvm_object

Class Declaration

```
virtual class uvm_object extends uvm_void
```

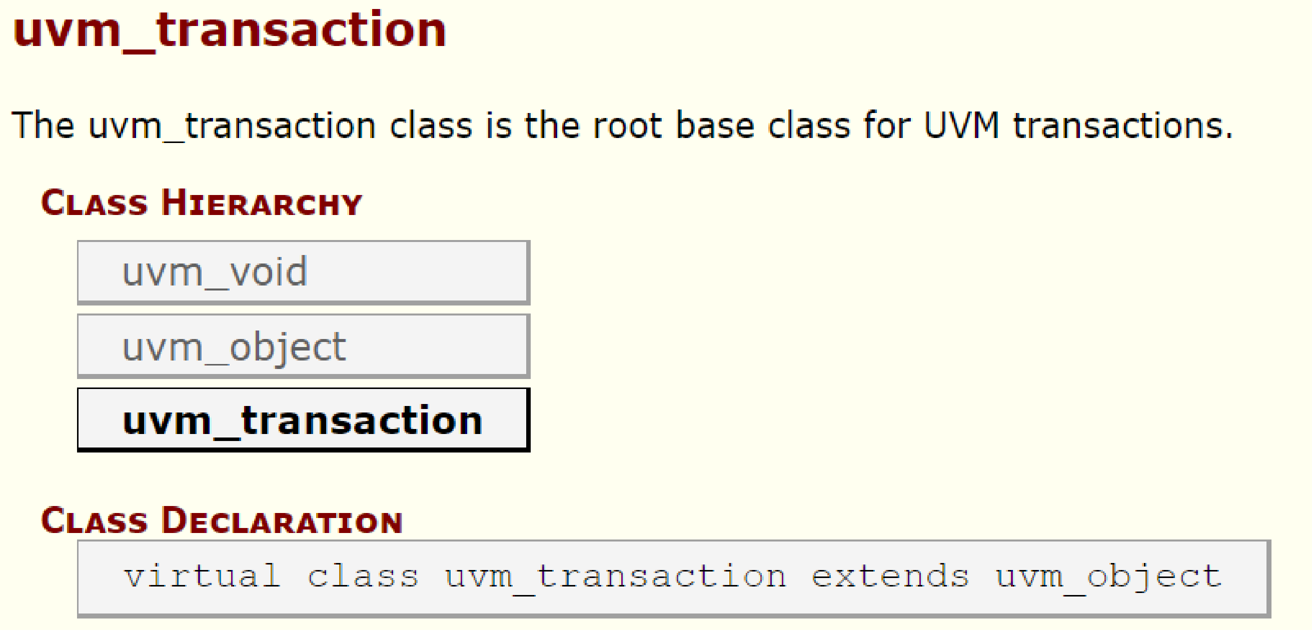
UVM Class Hierarchy

UVM report object

- Provides reporting functionality for UVM.

UVM transaction

- Used for generating stimulus and its analysis.
- They are transient in nature.



uvm_transaction

The uvm_transaction class is the root base class for UVM transactions.

Class Hierarchy

uvm_void
uvm_object
uvm_transaction

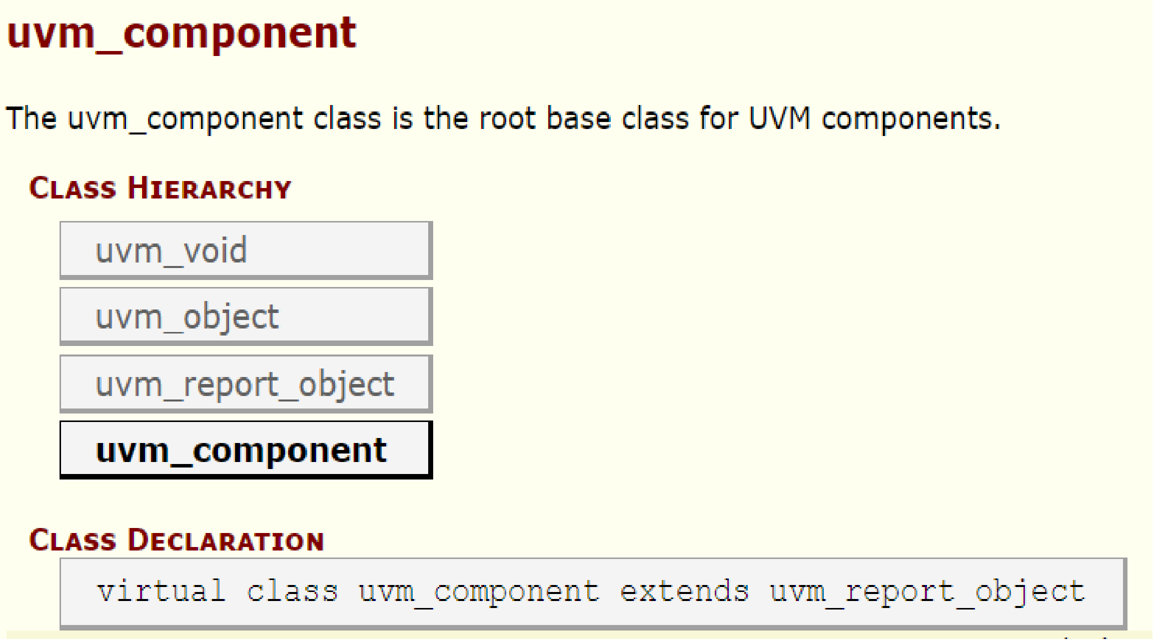
Class Declaration

```
virtual class uvm_transaction extends uvm_object
```

UVM Class Hierarchy

UVM component

- The uvm_component class is the root base class for all UVM components. Components are static objects that exist throughout simulation.



uvm_component

The uvm_component class is the root base class for UVM components.

Class Hierarchy

uvm_void
uvm_object
uvm_report_object
uvm_component

Class Declaration

```
virtual class uvm_component extends uvm_report_object
```

- Class constructor new() needs the instance name and handle to its parent.
- All classes derived from uvm_component must call super.new(name,parent).

UVM Class Hierarchy

UVM component provides many features like:

- Hierarchy: Provides methods for searching and traversing component hierarchy. (get_parent, get_full_name)

- 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.

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

UVM Class Hierarchy

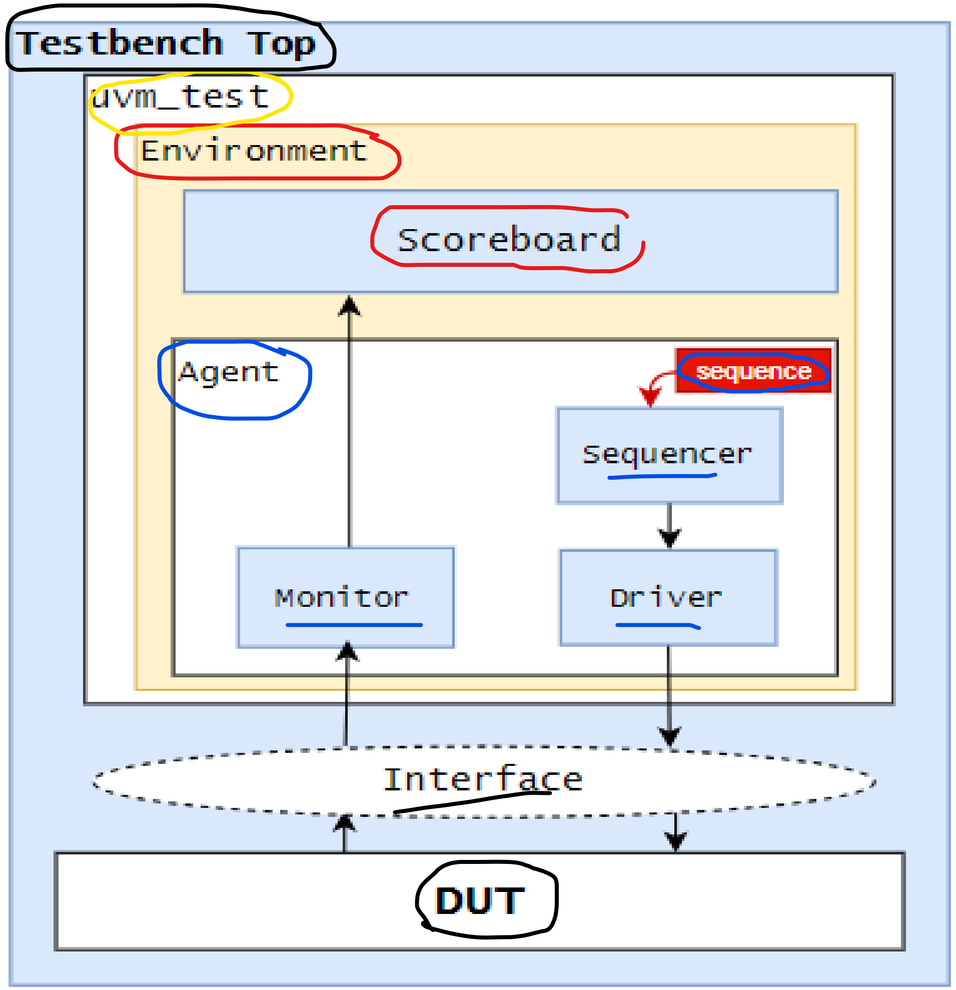
UVM component provides many features like:

- Objection: Provides an interface to the uvm_objection mechanism.
- Configuration: UVM provides a configuration database that allows users to store and retrieve configuration information for testbench components
- 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.

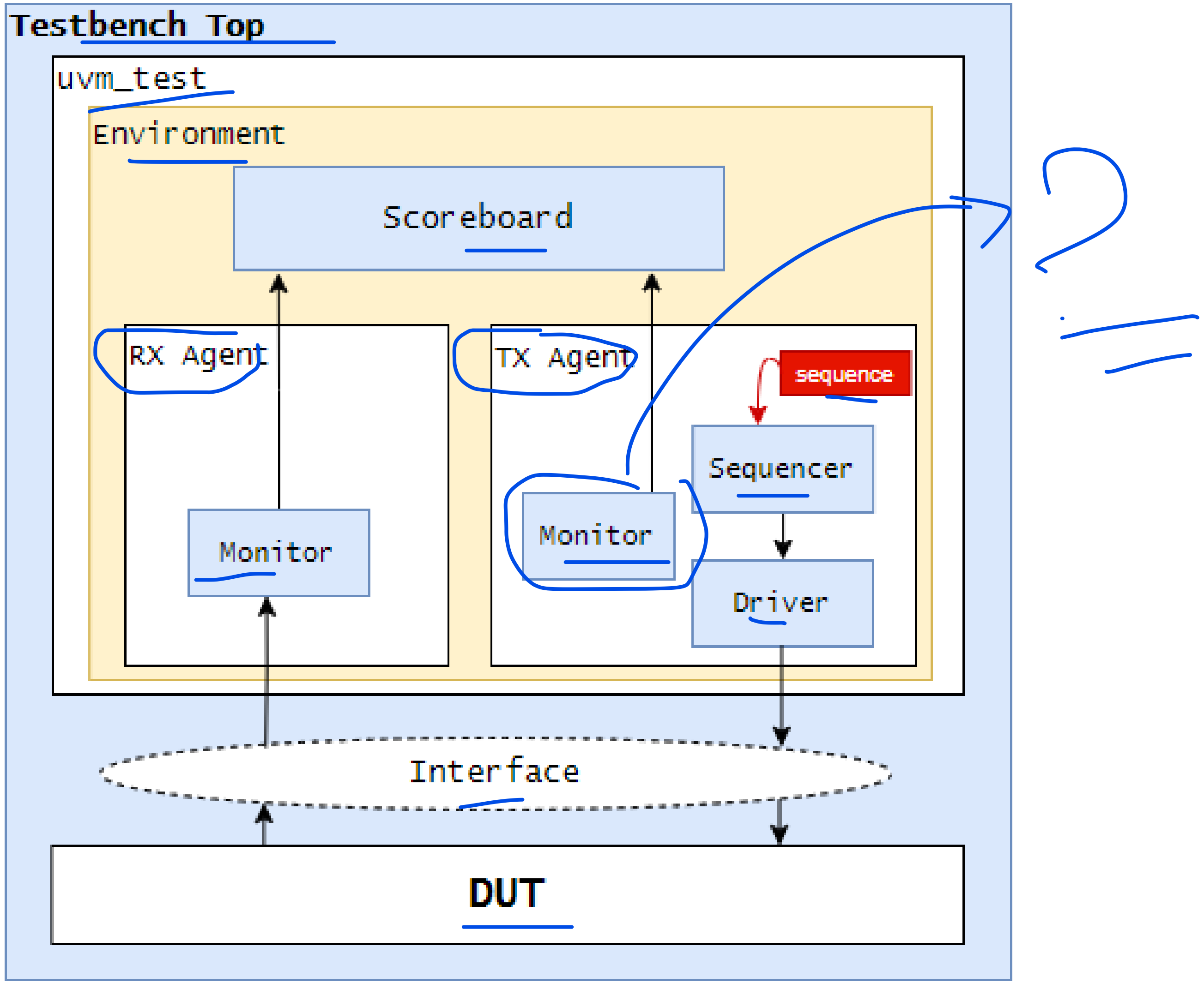
Default constructor for uvm_component has two arguments: name and parent.
Default constructor for uvm_object has a single argument: name

UVM Testbench Hierarchy

- The UVM employs a layered, object-oriented testbench hierarchy that allows "separation of concerns" among the various team members.



UVM Testbench Hierarchy



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.
- UVM testbench top is also used to trigger a test using run_test() call.

UVM Testbench Top

generate clk, rst

in Stance of intrFace

DUT

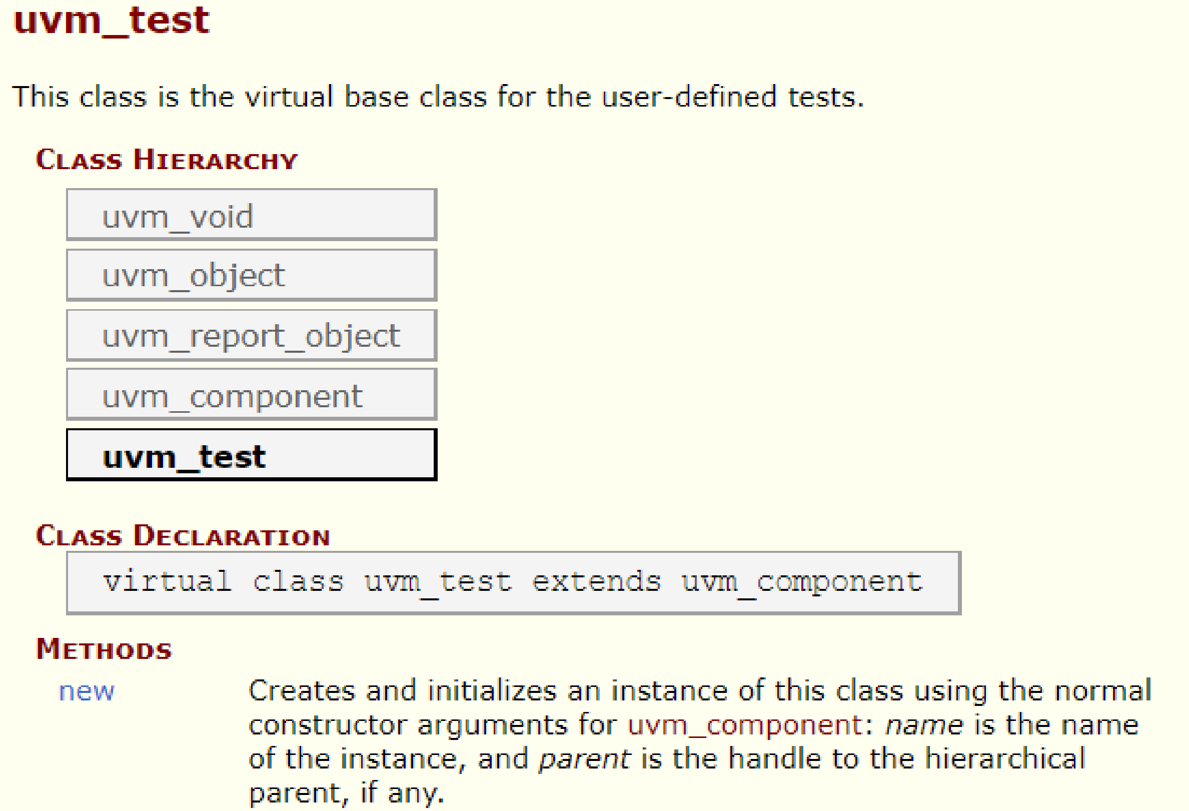
```
`include "uvm_macros.svh"
import uvm_pkg::*;

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
```

put intrface in DB

UVM Test

- ○ The top-level UVM Component in the UVM Test bench.
- ○ Instantiates the top-level environment and configures it.
- ○ Implement run phase to start sequences on required sequencers with raise/drop objection callbacks.



uvm_test

This class is the virtual base class for the user-defined tests.

Class Hierarchy

- uvm_void
- uvm_object
- uvm_report_object
- uvm_component
- **uvm_test**

Class Declaration

```
virtual class uvm_test 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.

UVM Test

```
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
```

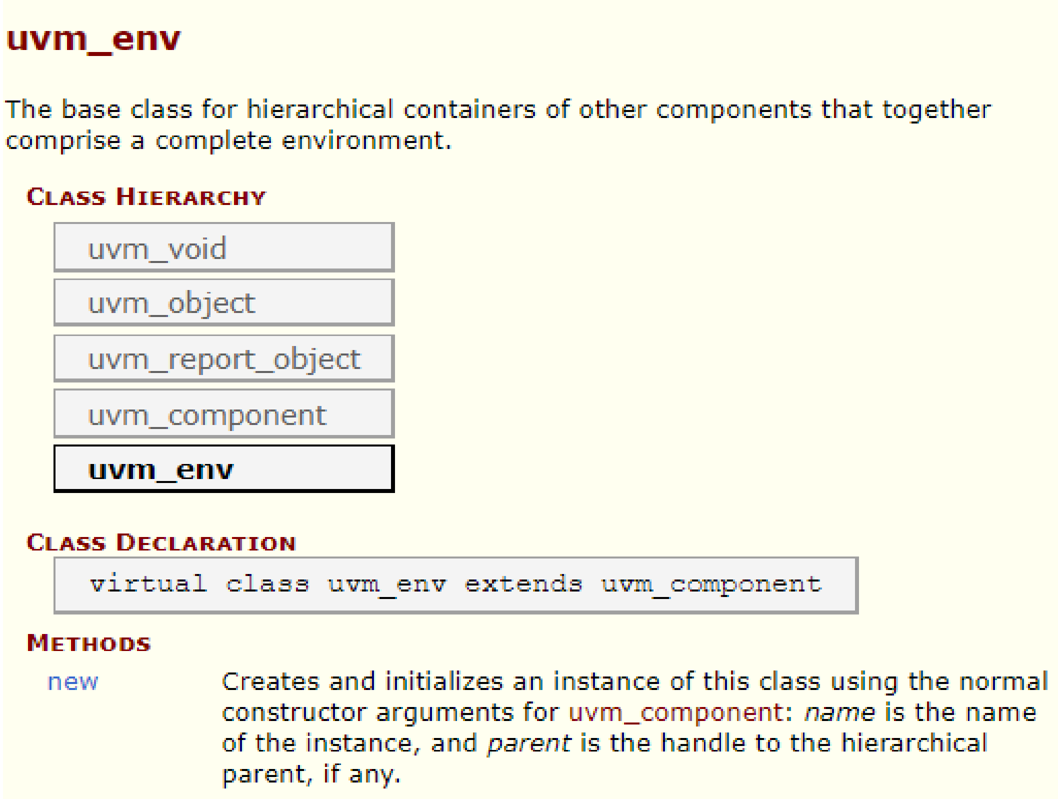
کلاس

```
  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
```

UVM Environment

○ The UVM Environment is a higher-level verification and hierarchical component that groups together other verification components that are interrelated.

○The top-level UVM Environment encapsulates all the verification components targeting the DUT like UVM Agents, UVM Scoreboards, or even other UVM Environments.



uvm_env

The base class for hierarchical containers of other components that together comprise a complete environment.

Class Hierarchy

uvm_void
uvm_object
uvm_report_object
uvm_component
uvm_env

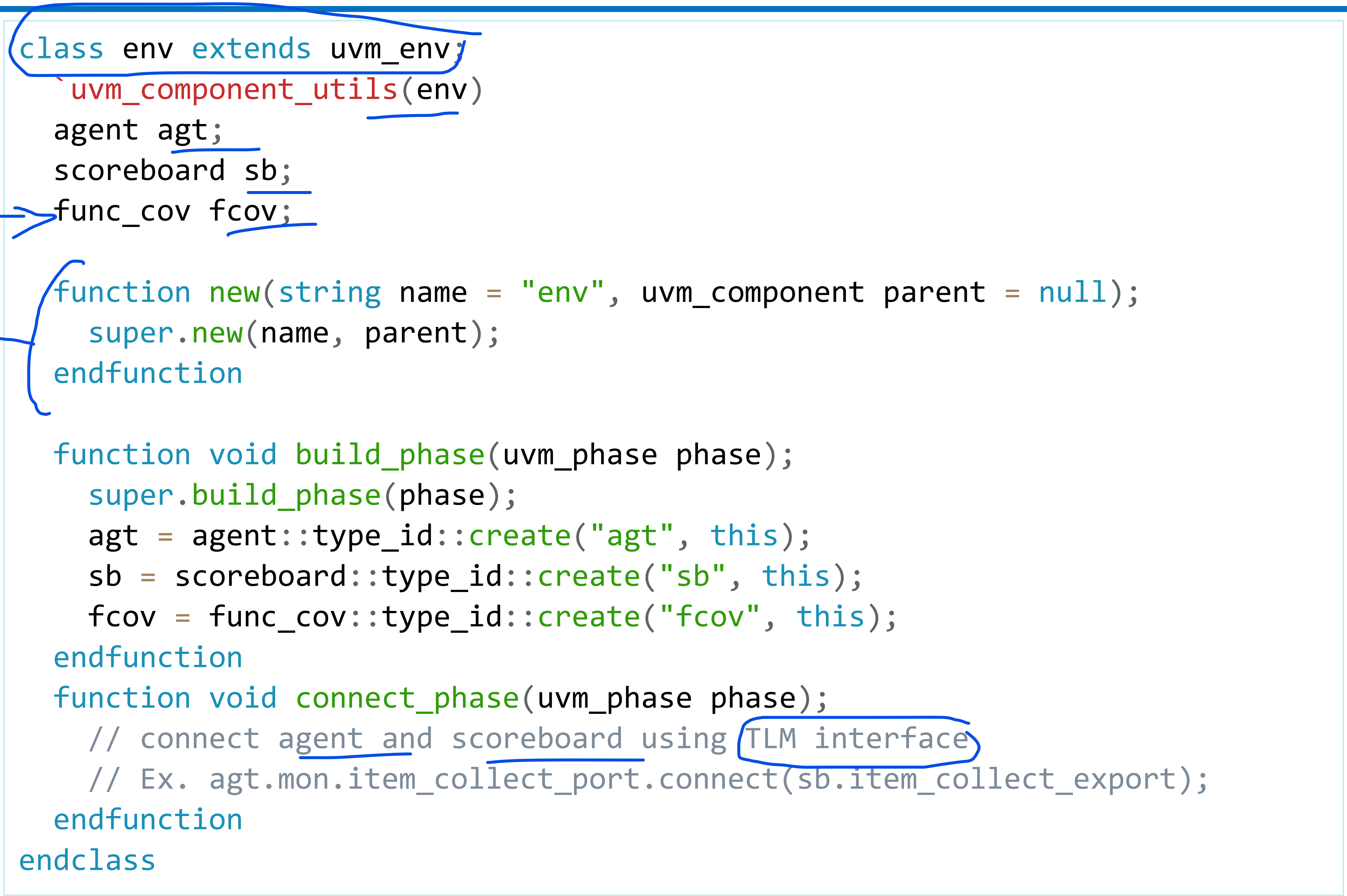
Class Declaration

```
virtual class uvm_env 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.

UVM Environment



```
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
```

for
factory

UVM Agent

- The UVM agent is a hierarchical component that contains UVM Sequencer, UVM Driver, and UVM Monitor that are dealing with a specific DUT interface.

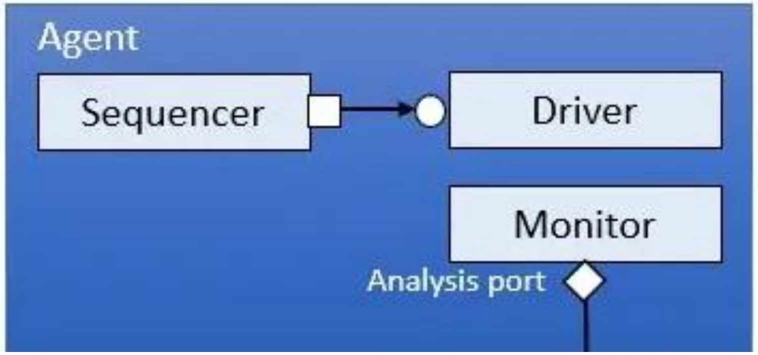
- UVM Agents might include other components, like coverage collectors, protocol checkers, a TLM model, etc.

- The UVM environment may contain more than one agent. The agent can initiate the transactions to the DUT or react to the transaction requests.

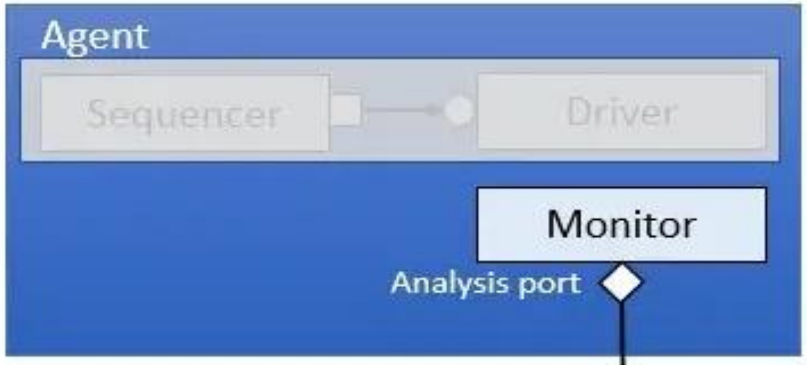
- There are two types of agents:
  1. Active
  2. Passive

UVM Agent

- ○ Active agent stimulates the DUT by driving transactions and monitors the device. It instantiates all three components driver, monitor, and sequencer.



- ○ 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.



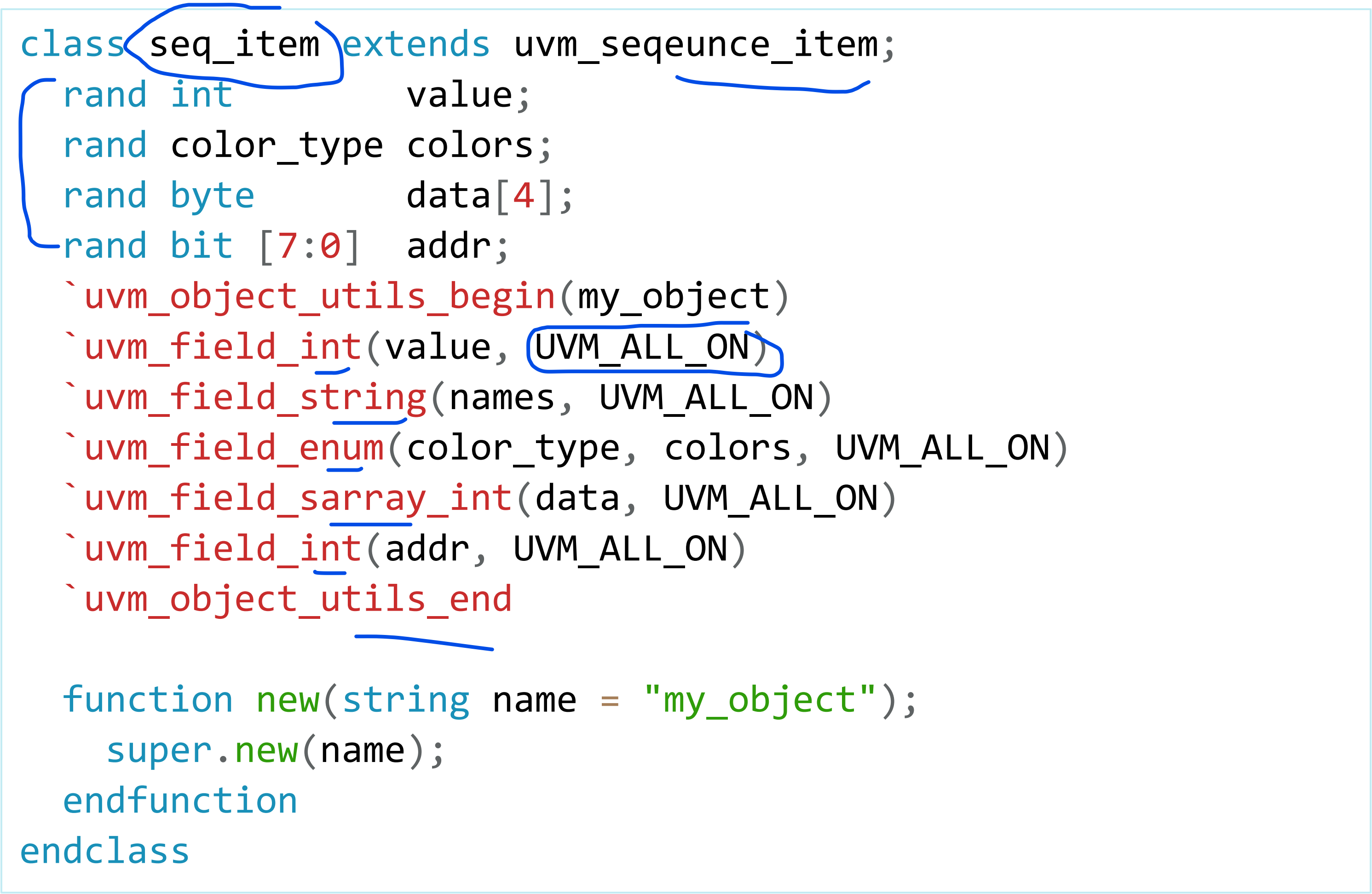
UVM Agent

```
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_ite
m_export);
  endfunction
endclass
```

UVM Sequence Items

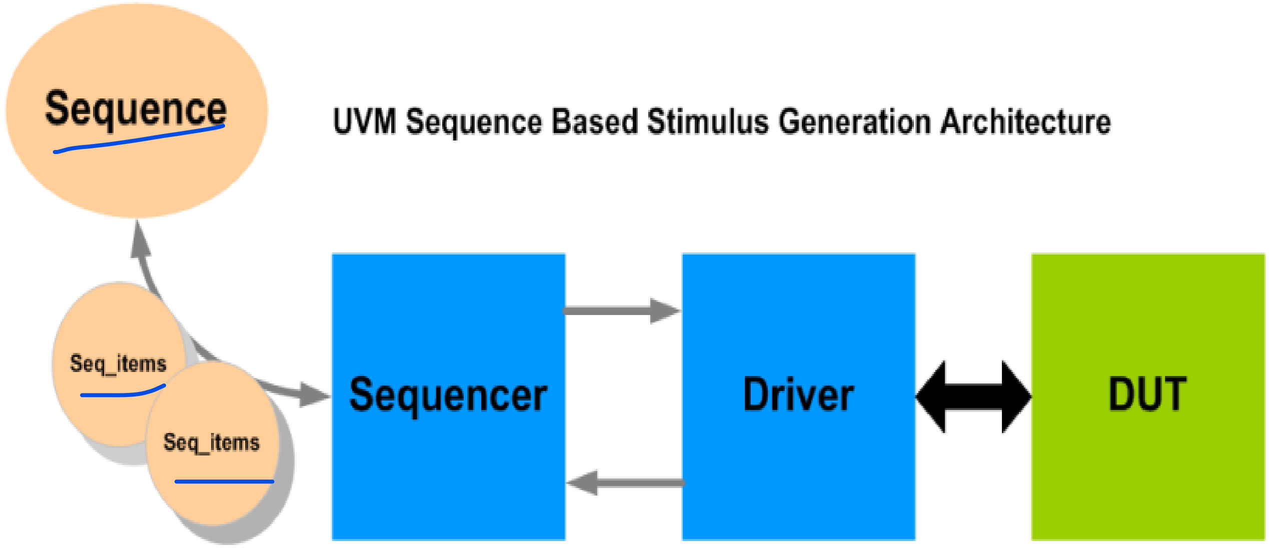
○ Generate stimulus and has control capabilities for the sequence-sequencer mechanism.



```
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
```

UVM Sequence

- ○ UVM sequence is a container that holds data items (uvm_sequence_items) which are sent to the driver via the sequencer.
- ○ Can be transient or persistent.
- ○ To operate, each UVM Sequence is eventually bound to a UVM Sequencer.
- ○ Multiple UVM Sequence instances can be bound to the same UVM Sequencer



UVM Sequence Based Stimulus Generation Architecture

UVM Sequence

```
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
```

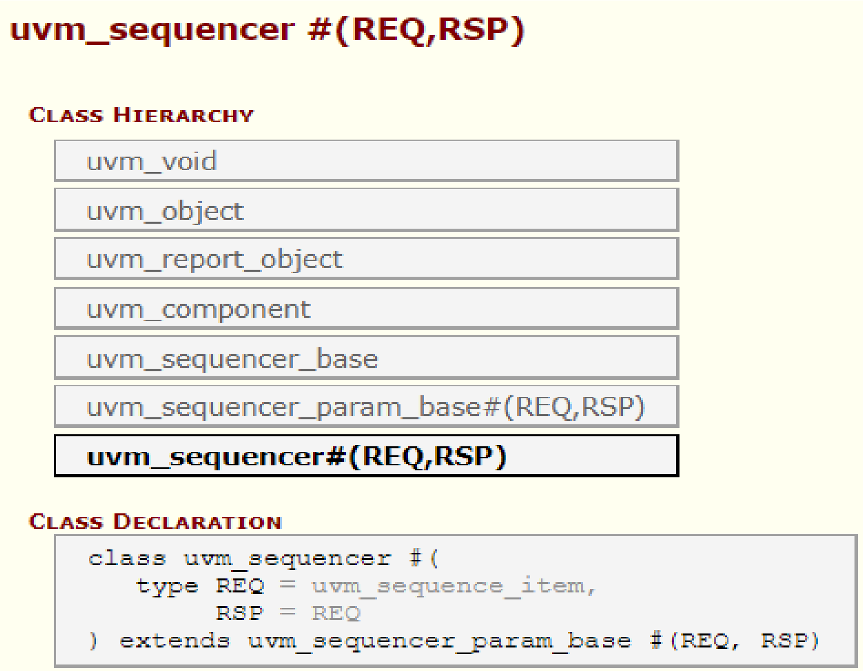
```
task body();
  `uvm_do(seq1); // calling seq1
  `uvm_do(seq2); // calling seq2
Endtask
```

```
task body();
  `uvm_create(req);
  assert(req.randomize());
  `uvm_send(req);
endtask
```

```
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
```

UVM Sequencer

- ○ The UVM sequencer behaves as an arbiter for controlling transaction flow from multiple stimulus sequences.
- ○ Also controls the flow of UVM Sequence Items generated by one or more UVM Sequences.
- ○ The uvm_sequencer class is a parameterized class of type REQ sequence_item and RSP sequence item. RSP sequence item is optional.



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)
```

UVM Sequencer

```
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 Driver

- The UVM Driver receives the sequence Item transactions from the UVM Sequencer and applies it on the DUT Interface.

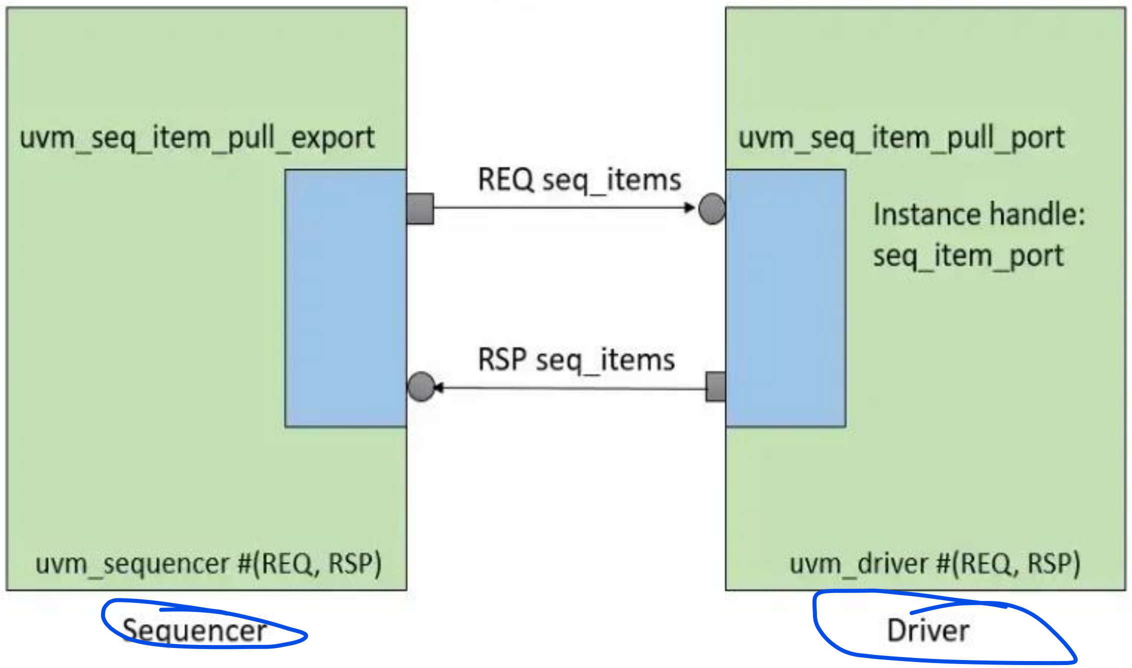
- The uvm_driver class is a parameterized class of type REQ sequence_item and RSP sequence item. RSP sequence item is optional.

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

UVM Driver

- The driver has uvm_seq_item_pull_port which is connected with uvm_seq_item_pull_export of the associated sequencer.

- 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.



UVM Driver

```
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
```

check
vir
interface

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

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

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.

○ 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 monitor calls analysis_port.write(), it basically cycles through the list and calls the write() method of each connected export.

UVM Monitor

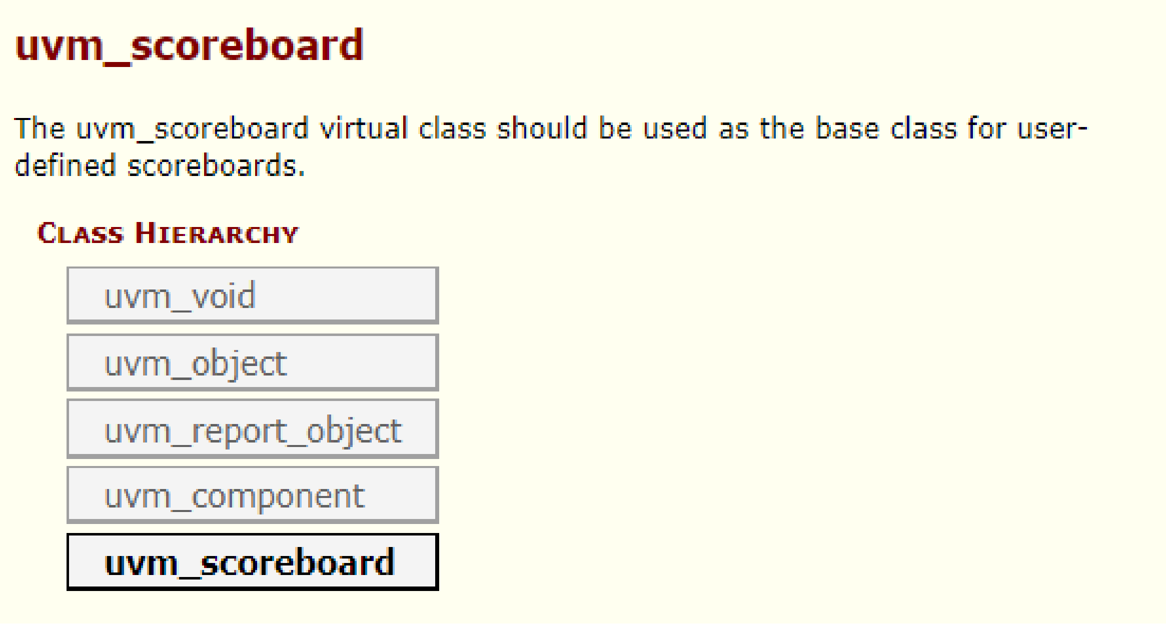
```
class monitor extends uvm_monitor;
  // declaration for the virtual interface, analysis port, and
monitor sequence item.
  virtual add_if vif;
  uvm_analysis_port #(seq_item) item_collect_port;
  seq_item mon_item;
  `uvm_component_utils(monitor)

  // 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
```

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

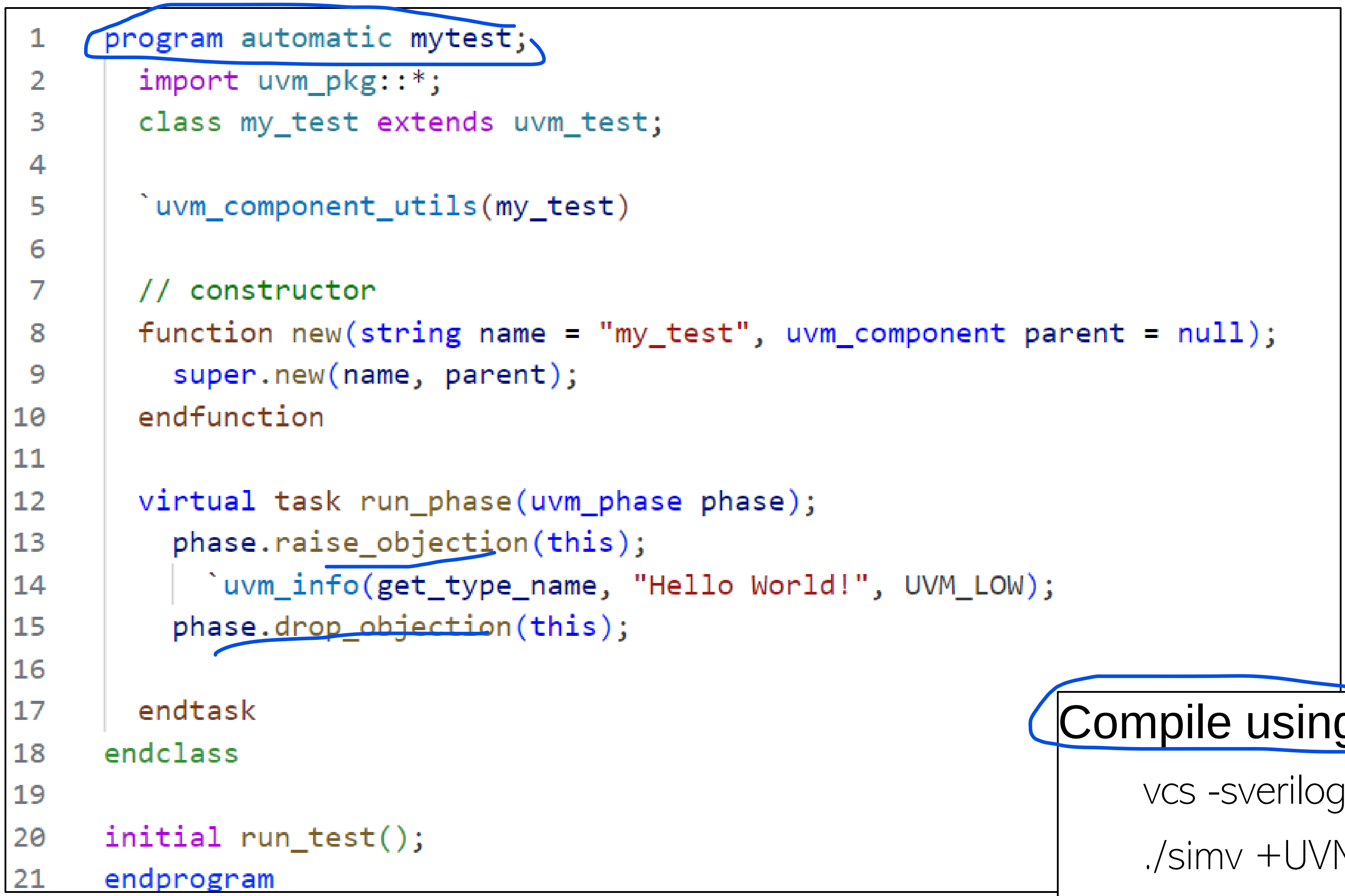
uvm_void
uvm_object
uvm_report_object
uvm_component
uvm_scoreboard

UVM Scoreboard

```
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
```

Try Building a Simple UVM Test



```
program automatic mytest;
  import uvm_pkg::*;
  class my_test extends uvm_test;

  `uvm_component_utils(my_test)

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

  virtual task run_phase(uvm_phase phase);
    phase.raise_objection(this);
      `uvm_info(get_type_name, "Hello World!", UVM_LOW);
    phase.drop_objection(this);

  endtask
endclass

initial run_test();
endprogram
```

Compile using:

```
vcs -sverilog -ntb_opts uvm-1.2 mytest.sv
./simv +UVM_TESTNAME=my_test
```

Thank you

CENTER OF NANOELECTRONICS & DEVICES