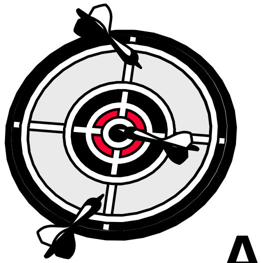
Agenda: Day 2

DAY 2

5	UVM Configuration & Factory
6	UVM Component Communication
7	UVM Scoreboard & Coverage
8	UVM Callback

Unit Objectives



After completing this unit, you should be able to:

- Embed UVM callback methods
- Build *façade* UVM callback classes
- Implement UVM callback to inject errors
- Implement UVM callback to implement coverage

Changing Behavior of Components

How to enable adding/modifying operation of a component?

- **One method: embed simple callbacks**

```
class driver extends uvm_driver #(packet);
  // utils macro and constructor not shown
  virtual task run_phase(uvm_phase phase);
    forever begin
      seq_item_port.get_next_item(req);
      pre_send(req);    // simple callback
      send(req);
      post_send(req);  // simple callback
      seq_item_port.item_done();
    end
  endtask
  virtual task send(packet tr); ...; endtask
  virtual task pre_send(packet tr); endtask   // required for simple callback
  virtual task post_send(packet tr); endtask // required for simple callback
endclass
```

Embed simple no-op methods before and after major operation

Simple callback method are no-op methods of the class

Implementing Simple Callback Operations

- **Simple callback requires one to extend from existing component class**

```
class driver_new extends driver;
  virtual task pre_send(…); …
  virtual task send(…); …
  virtual task post_send(…); …
endclass
```

In the derived class, one can implement the callback methods

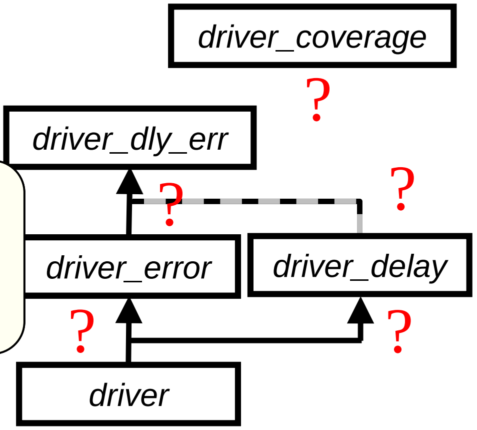
- **This works well for making same change for all tests**
- **But, causes problems for testcase only changes**
  - Multiple extensions can cause unstable OOP hierarchy
    - How many versions of drivers to maintain?
    - How to add multiple extensions?

```
class driver_error extends driver;
```

```
class driver_delay extends driver;
```

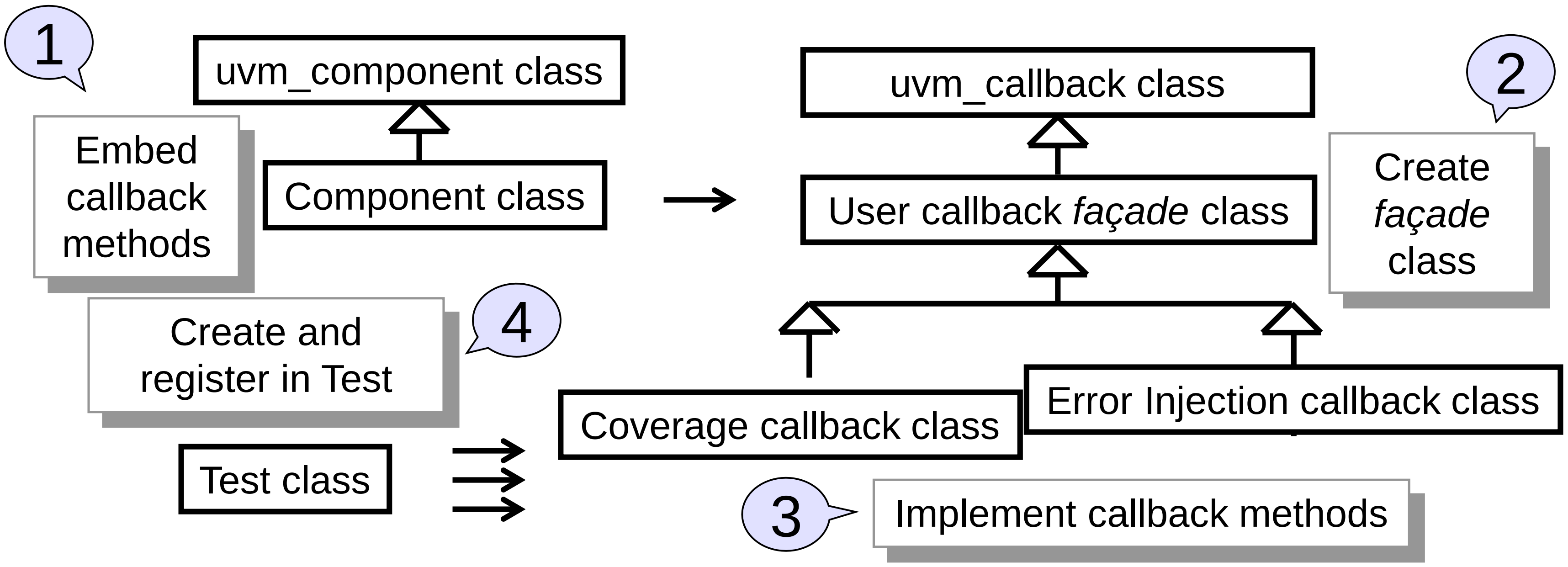
```
class driver_dly_err extends driver_error;
```

What to extend from if multiple requirements for a test?



Implementing UVM Callbacks

- **Use UVM callbacks to add new capabilities, without creating huge OOP hierarchy**
- **Four steps:**
  - Embed UVM callback methods in components
  - Create a *façade* UVM callback class
  - Develop UVM callback classes extending from *façade* callback class
  - Create and register UVM callback objects in environment



Step 1: Embed Callback Methods

- **Typically before and/or after major operation**

```
class driver extends uvm_driver #(packet);
  `uvm_register_cb(driver, driver_callback)
  // utils macro and constructor not shown
  virtual task run_phase(uvm_phase phase);
    forever begin
      seq_item_port.get_next_item(req);

      `uvm_do_callbacks(driver, driver_callback, pre_send(this, req));
      send(req);


      `uvm_do_callbacks(driver, driver_callback, post_send(this, req));
      seq_item_port.item_done();
    end
  endtask
endclass
```

1a. Register UVM callback with component

1b. Embed UVM callback methods with uvm_do_callbacks macro

UVM callback class name
User must create (see next slide)

Component class name

UVM callback method
User must embed in callback class (see next slide)

Step 2: Declare the *façade* Class

- **Create *façade* class called in `uvm_do_callbacks` macro**
  - Typically declared in same file as the component
  - All methods must be declared as virtual
  - Leave the body of methods empty

2. Create callback *façade* class

```
typedef class driver;
class driver_callback extends uvm_callback;      // utils macro not needed
  function new(string name = "driver_callback");
    super.new(name);
  endfunction
  virtual task pre_send(driver drv, packet tr); endtask
  virtual task post_send(driver drv, packet tr); endtask
endclass
```

Empty body: noop

Argument types must match types in `uvm_do_callbacks() macro

Step 3: Implement Callback: Error

- **Create error class by extending from *façade* class**
  - Embed error in callback method

3. Implement error callback

```
class driver_err_callback extends driver_callback;
  virtual task pre_send(driver drv, packet tr);
    tr.payload.delete();
  endtask
endclass
```

Step 4: Create and Register Callback Objects

- **Instantiate the callback object in test**
- **Construct and register callback object**

```
class test_driver_err extends test_base;
 // utils macro and constructor not shown
  driver_err_callback drv_err_cb;
 virtual function void connect_phase(uvm_phase phase);
  super.connect_phase(phase);
  drv_err_cb = new();
  uvm_callbacks #(driver, driver_callback)::add(env.agt.drv, drv_err_cb);
  uvm_callbacks #(driver, driver_callback)::display();
 endfunction
endclass
```

4a. Create callback objects

4b. Register callback objects

4c. Visually verify the registration (optional)

Driver Coverage Example

- **If there are no analysis ports in driver**
  - Callbacks can be the hooks for coverage also

```
typedef class driver;
class driver_callback extends uvm_callback;
  // constructor not shown
  virtual task pre_send(driver drv, packet tr); endtask
  virtual task post_send(driver drv, packet tr); endtask
endclass
class driver extends uvm_driver #(packet);
  `uvm_register_cb(driver, driver_callback)
  // utils macro and constructor not shown
  virtual task run_phase(uvm_phase phase);
    forever begin
      seq_item_port.get_next_item(req);
      `uvm_do_callbacks(driver, driver_callback, pre_send(this, req));
      send(req);
      `uvm_do_callbacks(driver, driver_callback, post_send(this, req));
      seq_item_port.item_done();
    end
  endtask
endclass
```

Implement Coverage via Callback

- **Create coverage class by extending from *façade* class**
  - Define covergroup in coverage class
  - Construct covergroup in class constructor
  - Sample coverage in callback method

3a. Extend *façade* class

```
class driver_cov_callback extends driver_callback;
  covergroup drv_cov with function sample(packet pkt);
    coverpoint pkt.sa; coverpoint pkt.da;
    cross pkt.sa, pkt.da;
  endgroup
  function new();
    drv_cov = new();
  endfunction
  virtual task post_send(driver drv, packet tr);
    drv_cov.sample(tr);
  endtask
endclass
```

3b. Implement coverage

Create and Register Callback Objects

- **Instantiate the callback in Environment**
- **Construct and register callback object in connect phase**

```
class test_driver_cov extends test_base;
 // utils macro and constructor not shown
 driver_cov_callback drv_cov_cb;
 virtual function void connect_phase(uvm_phase phase);
  super.connect_phase(phase);

  drv_cov_cb = new();
  uvm_callbacks #(driver,driver_callback)::add(env.agt.drv, drv_cov_cb);
//  uvm_callbacks #(driver, driver_callback)::add(null, drv_cov_cb);
 endfunction
endclass
```

4a. Create callback objects

4b. Register callback objects

Alternative: Register callback objects to all driver objects

User Callback Debug

- **Run-time switch:**
  - +UVM_CB_TRACE_ON

```
VCD+ Writer F-2011.12 Copyright (c) 1991-2011 by Synopsys Inc.
UVM_INFO /global/apps5/vcs_2011.12/etc/uvm-
1.1/base/uvm_callback.svh(631) @ 0: reporter [UVMCB_TRC] Add
(UVM_APPEND) typewide callback uvm_report_catcher for type  : callback
uvm_report_catcher (uvm_callback@465)
UVM_INFO @ 0.0ns: reporter [RNTST] Running test test_base...
UVM_INFO reset_agent.sv(28) @ 0.0ns: uvm_test_top.env.r_agt [RSTCFG]
Reset agent r_agt setting for is_active is: UVM_ACTIVE
UVM_INFO /global/apps5/vcs_2011.12/etc/uvm-
1.1/base/uvm_callback.svh(639) @ 0.0ns: reporter [UVMCB_TRC] Add
(UVM_APPEND) callback sb_callback to object uvm_test_top.env.sb  :
callback sb_callback (uvm_callback@7788)
```

What was appended

Where the callback object is appended

Sequence Simple Callback Methods

User should not call these methods directly. Instead, override in sequence definition

- **uvm_sequence::pre_start()   (task)**
  - called at the beginning of start() execution
- **uvm_sequence::pre_body()   (task)**
  - Called before sequence body execution
- **uvm_sequence::pre_do()   (task)**
  - called after sequencer::wait_for_grant() call and after sequencer has selected this sequence, but before the item is randomized
- **uvm_sequence::mid_do()   (function)**
  - called after sequence item randomized, but before it is sent to driver
- **uvm_sequence::post_do()   (function)**
  - called after the driver indicates item completion,using item_done/put
- **uvm_sequence::post_body()   (task)**
  - Called after sequence body execution
- **uvm_sequence::post_start()   (task)**
  - called at the end of start() execution

Unit Objectives Review

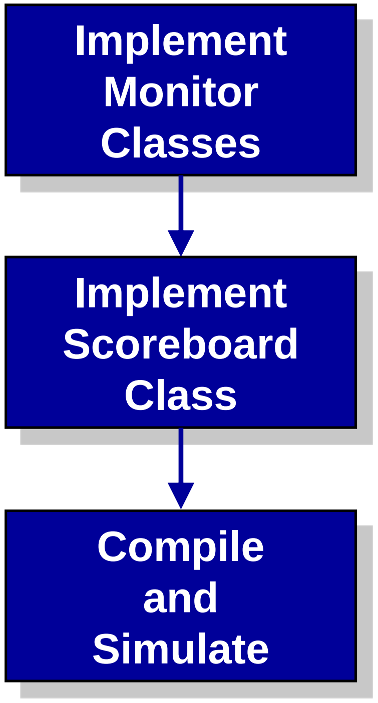
Having completed this unit, you should be able to:

- **Embed UVM callback methods**
- **Build *façade* UVM callback classes**
- **Implement UVM callback to inject errors**
- **Implement UVM callback to implement coverage**

Lab 4 Introduction

60 min

Implement monitors and scoreboard


Implement Monitor Classes
Implement Scoreboard Class
Compile and Simulate