Easier UVM – Functional Verification for Mainstream Designers

John Aynsley, Doulos

Easier UVM –
Functional Verification for Mainstream Designers

- *Introducing UVM*
- *Transactions and Components*
- *Sequencers and Drivers*
- *Configurations and the Factory*
- *What next?*

DOULOS

UVM

Easier UVM?

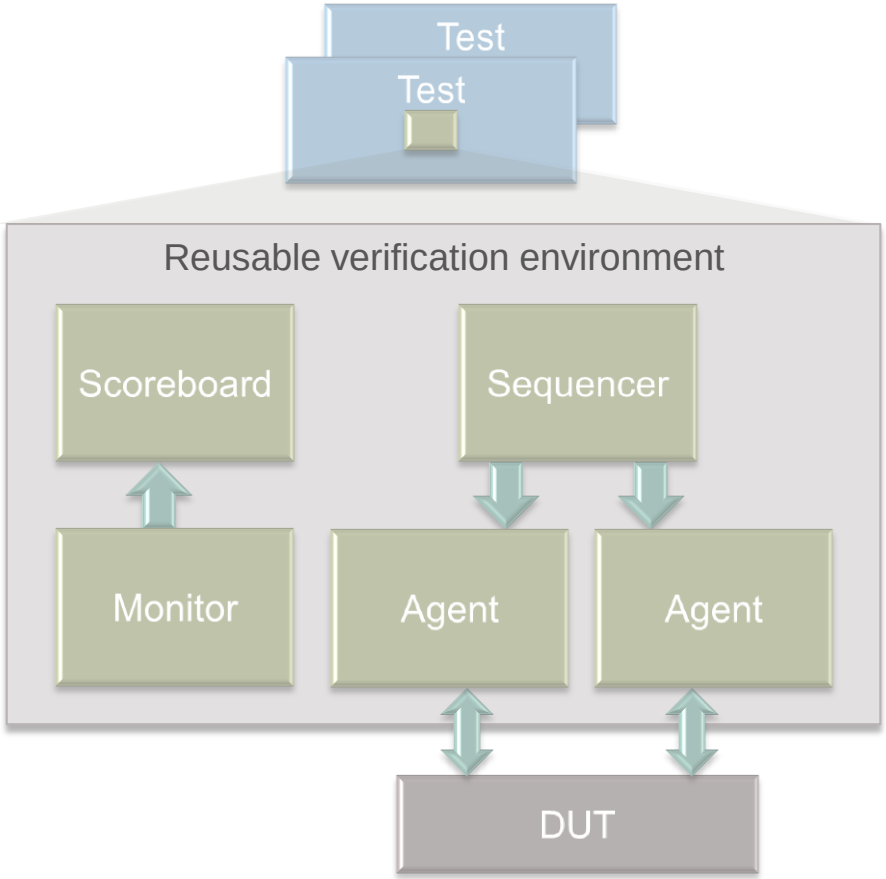
- Aimed at mainstream Verilog & VHDL users
- Goal = Reduce UVM to a set of simple concepts and coding idioms
- *Easier UVM* is UVM
- Use more features of UVM as you learn
- Learning UVM is still not easy...

Why UVM?

UVM = Universal Verification Methodology

The big wins are

- Verification quality
- Testbench reuse
- Knowhow reuse



UVM Highlights 1

- Constrained random verification
- Configurable, flexible, test benches

A standard approach – consistency and uniformity

- Open source SystemVerilog Base Class Library
- Supported by all the main vendors

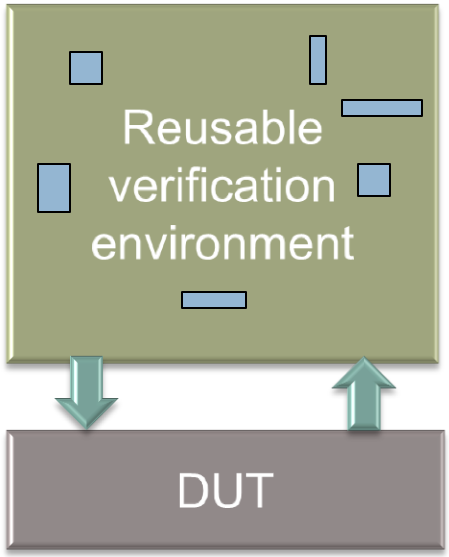
UVM Highlights 2

- Separation of tests from test bench
- Transaction-level communication (TLM)
- Layered sequential stimulus
- Message reporting
- Register layer

Test versus Testbench

Test writer

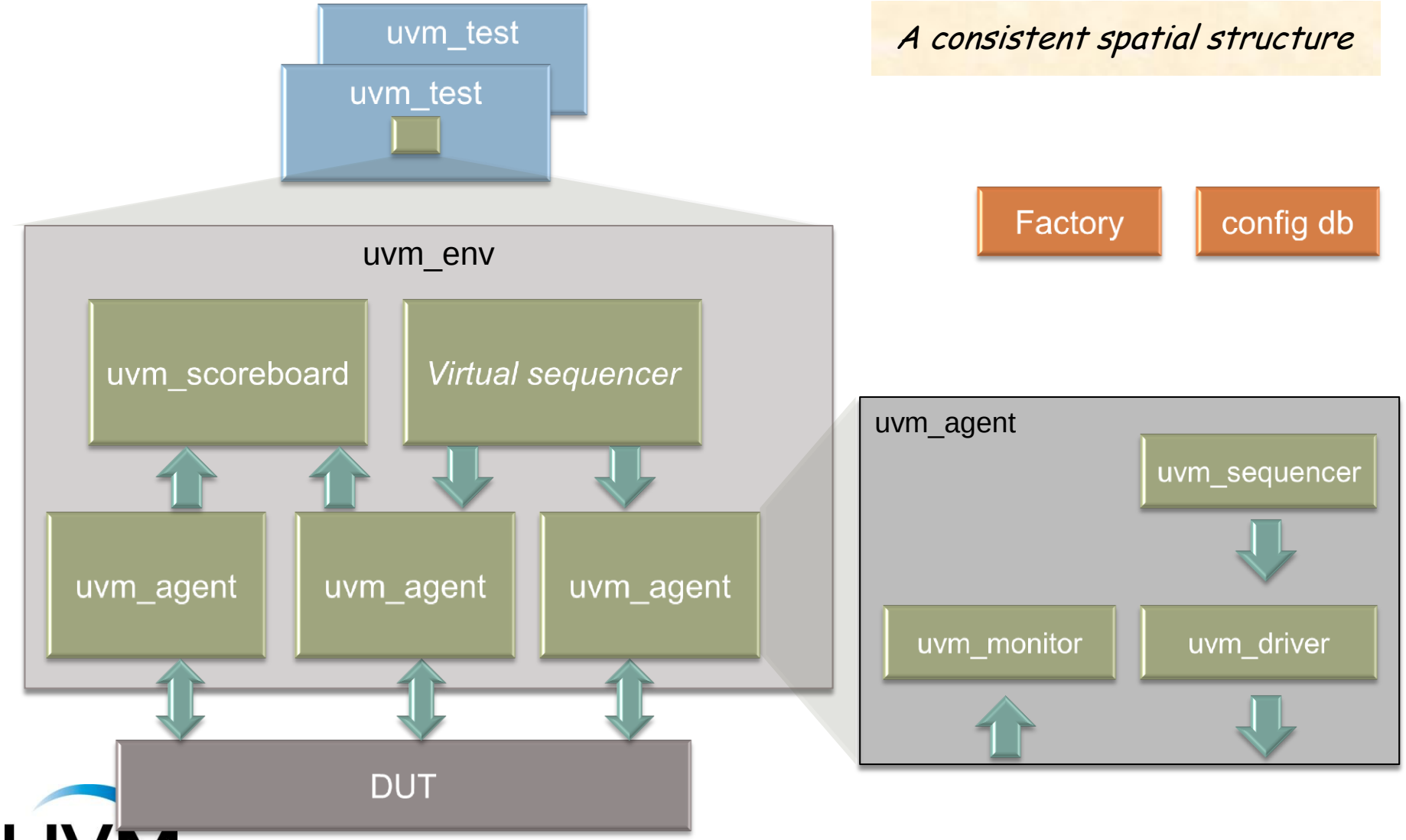
Verification specialist



Tests define differences

Allows for unanticipated changes

Testbench Structure



Quasi-static vs Dynamic Objects

Quasi-static vs Dynamic Objects

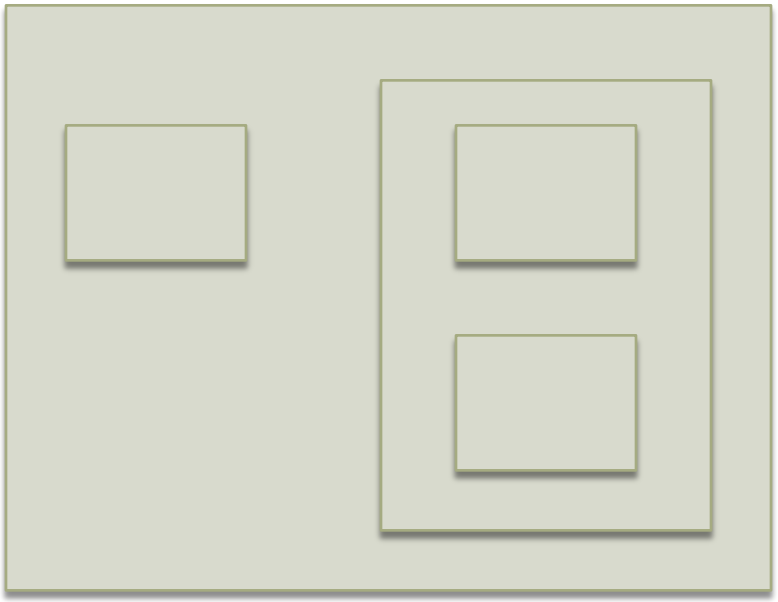
Structure

Component

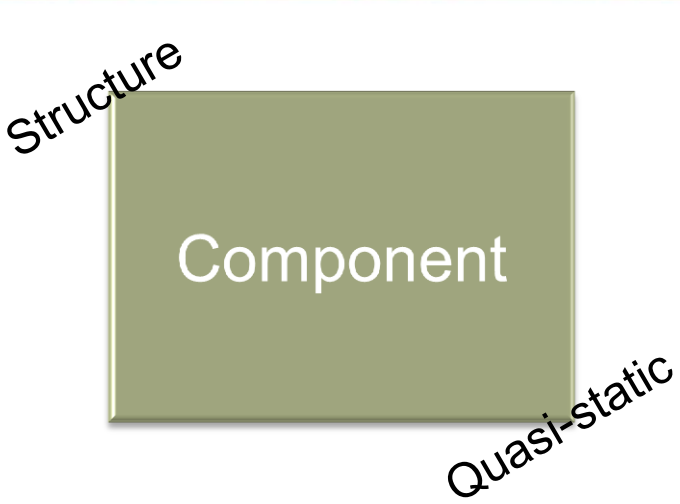
Quasi-static

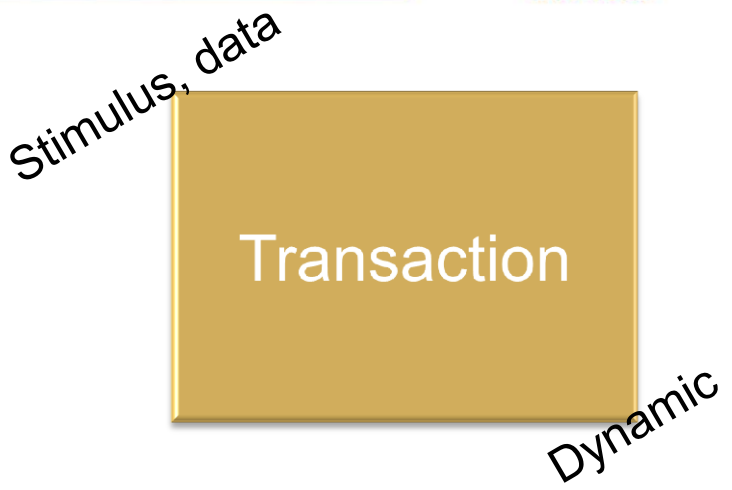
Stimulus, data

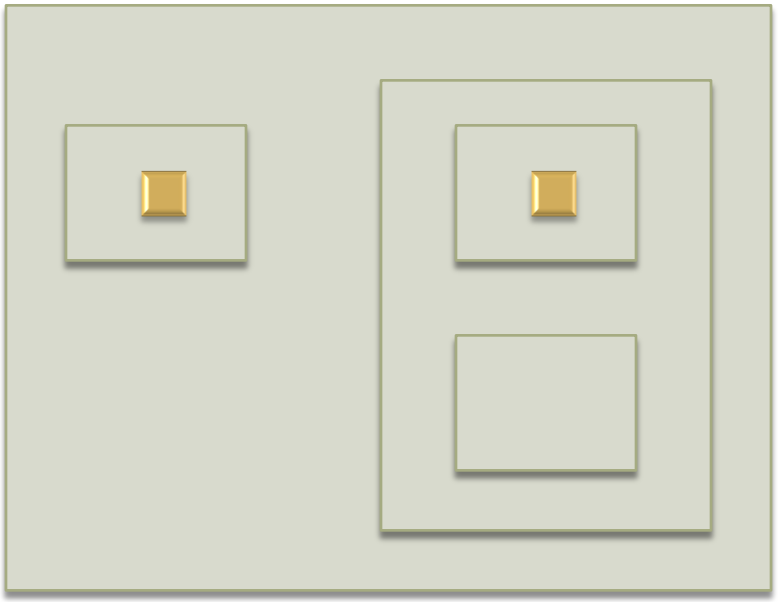
Transaction



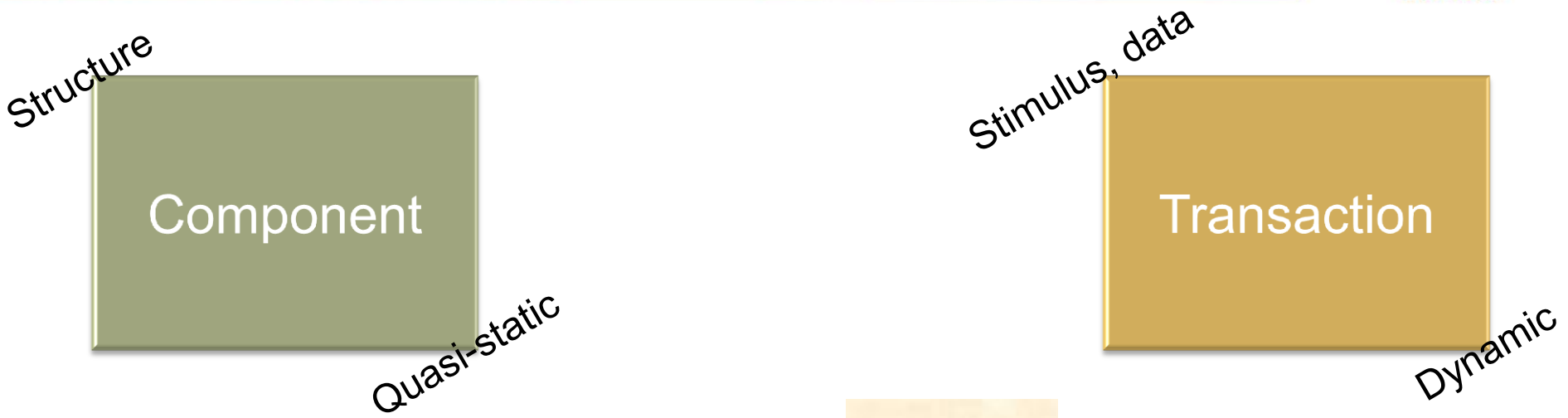
Quasi-static vs Dynamic Objects







Quasi-static vs Dynamic Objects



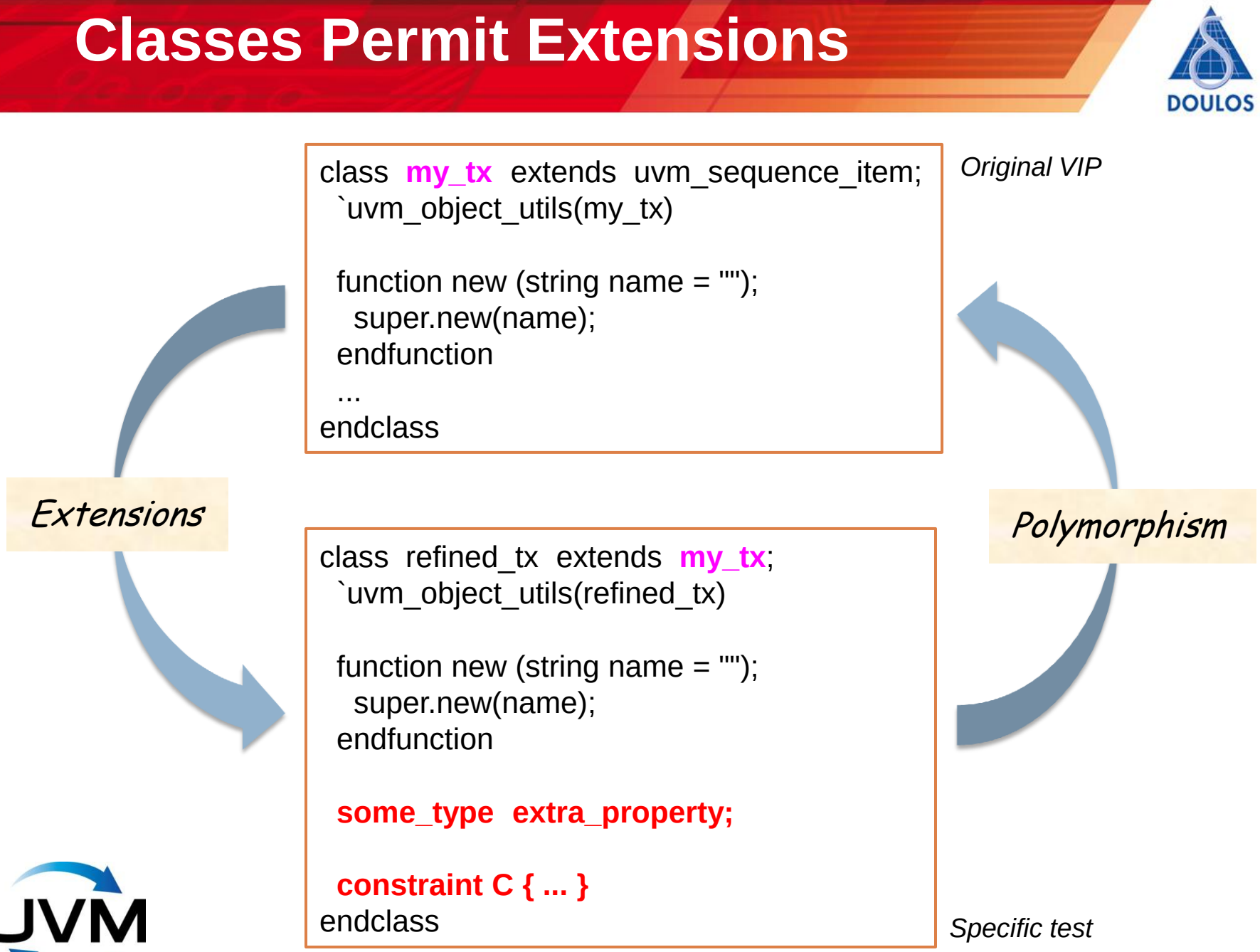
Pattern 1

```
class my_comp extends uvm_component;
  `uvm_component_utils(my_comp)

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

Pattern 2

```
class my_tx extends uvm_sequence_item;
  `uvm_object_utils(my_tx)

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

Classes Permit Extensions

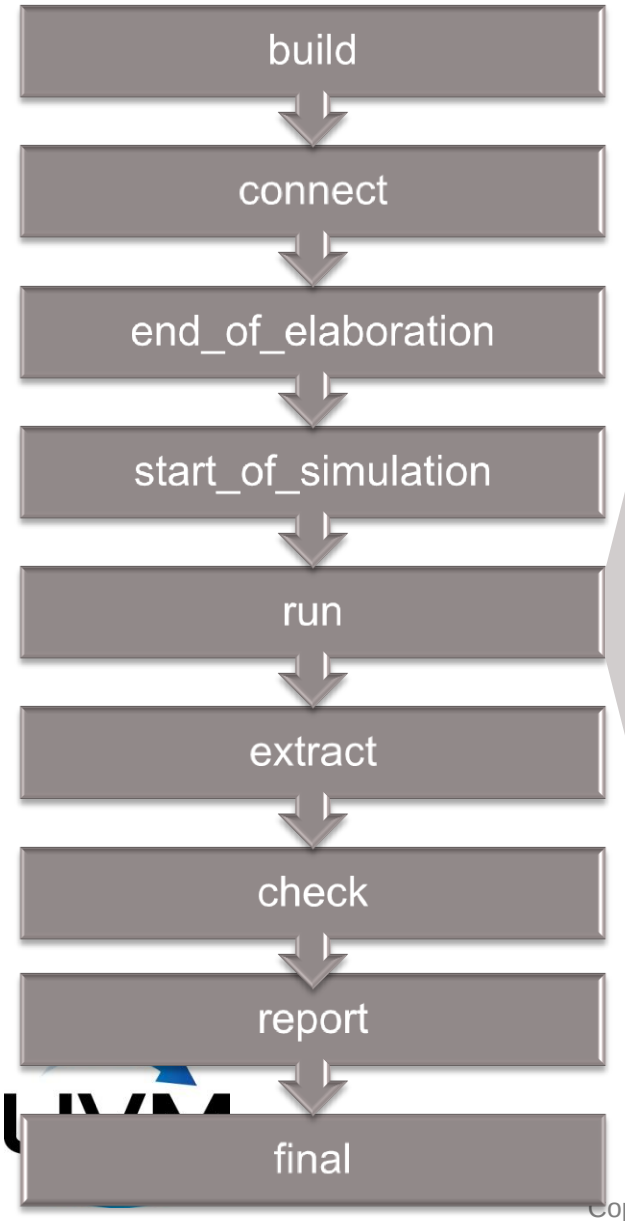


Transactions Methods

Pattern 2a

```
class my_tx extends uvm_sequence_item;
  `uvm_object_utils(my_tx)

  rand bit cmd;
  rand int addr;
  rand int data;

  constraint c_addr { addr >= 0; addr < 256; }
  constraint c_data { data >= 0; data < 256; }

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

  function string convert2string;
    ...

  function void do_copy( uvm_object rhs );
    ...

  function bit do_compare( uvm_object rhs, uvm_comparer comparer );
    ...
endclass
```

Execution Phases



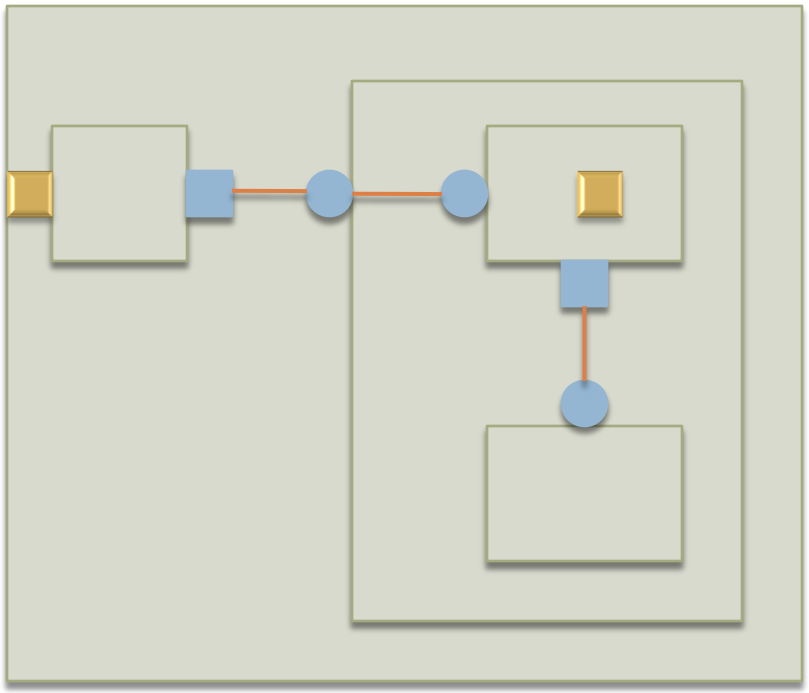
A consistent temporal structure

pre_reset
reset
post_reset

pre_configure
configure
post_configure

pre_main
main
post_main

pre_shutdown
shutdown
post_shutdown



Phase Methods of a Component

Pattern 1

```
class my_comp extends uvm_component;
  `uvm_component_utils(my_comp)

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

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

  function void connect_phase (uvm_phase phase);
    ...
  endfunction

  task run_phase (uvm_phase phase);
    ...
  endtask

  function void report_phase (uvm_phase phase);
  ...
endclass
```

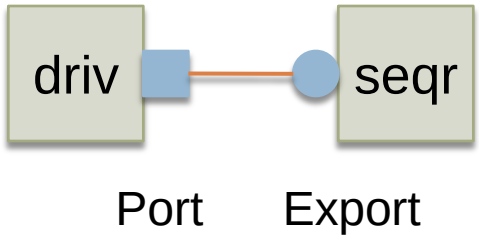
Build Phase

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

  my_driver          driv;
  my_sequencer  seqr;
  ...
  function  void  build_phase (uvm_phase phase);
    super.build_phase(phase);
    driv  = my_driver::type_id::create("driv", this);
    seqr = my_sequencer::type_id::create("seqr", this);
  endfunction
```

Factory-made components

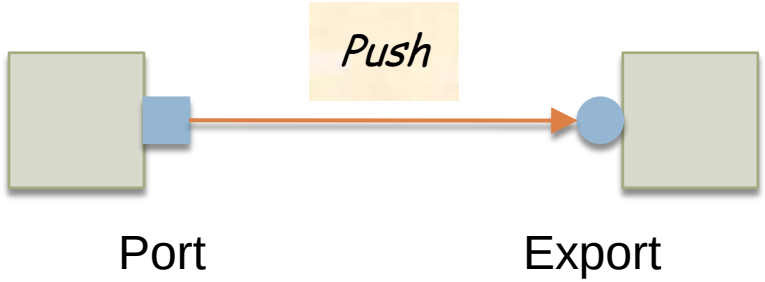
Connect Phase

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

  my_driver          driv;
  my_sequencer  seqr;
  ...
  function  void  build_phase (uvm_phase phase);
    super.build_phase(phase);
    driv  = my_driver::type_id::create("driv", this);
    seqr = my_sequencer::type_id::create("seqr", this);
  endfunction

  function  void  connect_phase (uvm_phase phase);
    driv.seq_item_port.connect( seqr.seq_item_export );
  endfunction
```



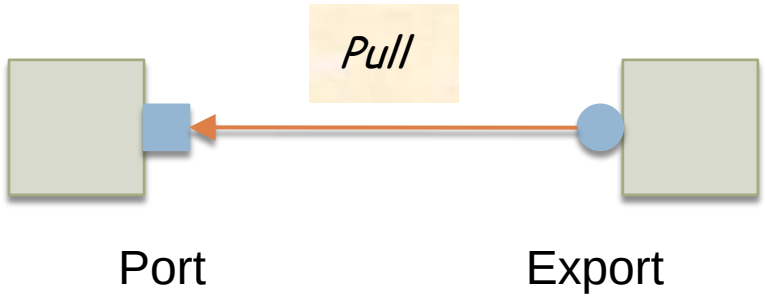
Transaction-Level Connections

port.put(tx);



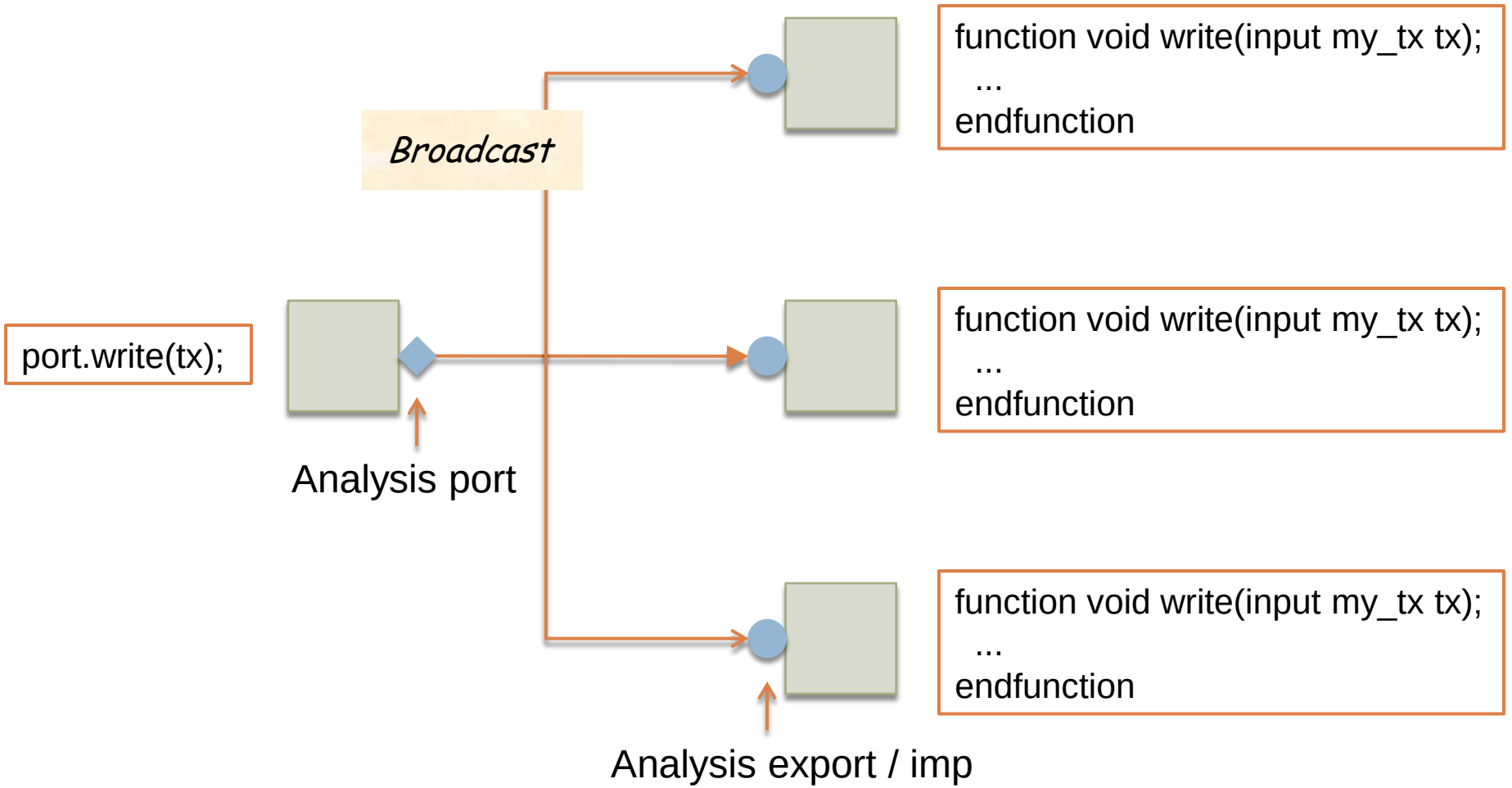

```
task put(input my_tx tx);
  ...
endtask
```

port.get(tx);



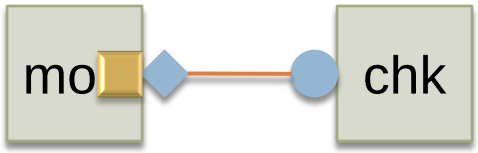

```
task get(output my_tx tx);
  ...
endtask
```

Analysis Ports

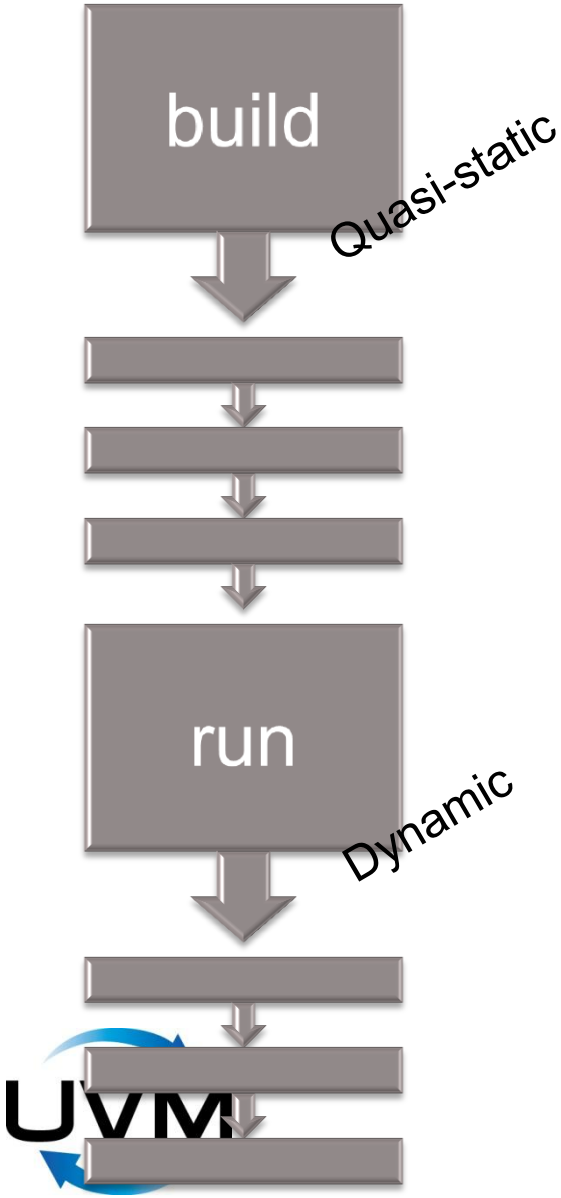


Run Phase

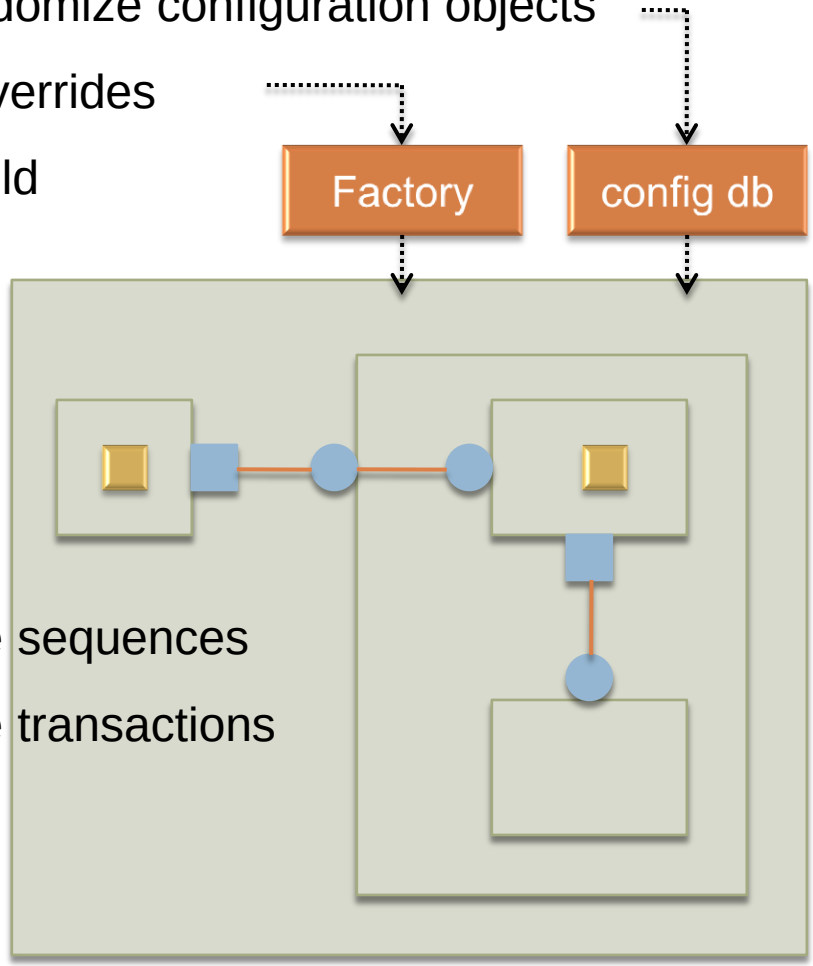
```
class  my_monitor  extends  uvm_component;
  `uvm_component_utils(my_monitor)

  uvm_analysis_port #(my_tx) port;
  ...
  function  void  build_phase (uvm_phase phase);
    super.build_phase(phase);
    port = new("port", this);
  endfunction

  task  run_phase (uvm_phase phase);
    my_tx  tx;
    tx = my_tx::type_id::create("tx");
    ...
    port.write(tx);
  endtask
```



Constrained Random Generation

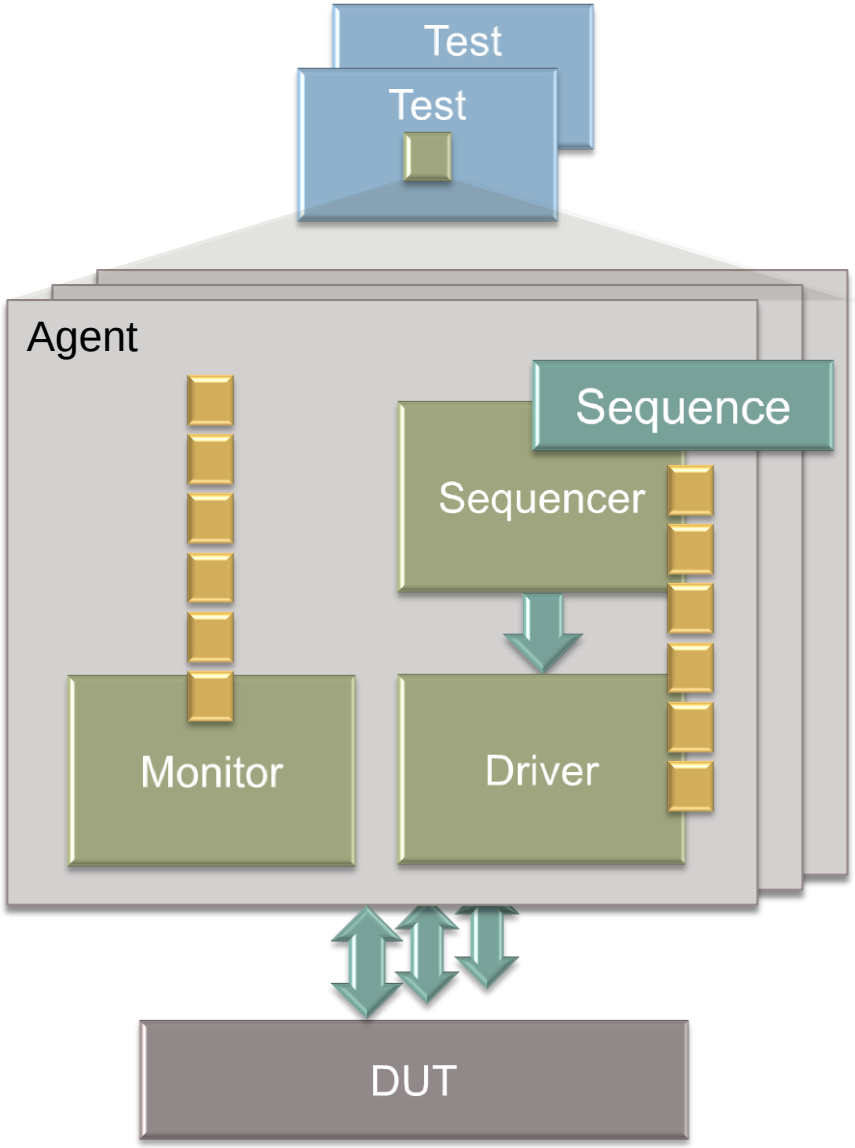


- Populate/randomize configuration objects
- Set factory overrides
- Top-down build



- Factory-made sequences
- Factory-made transactions
- Constraints

Agent Architecture

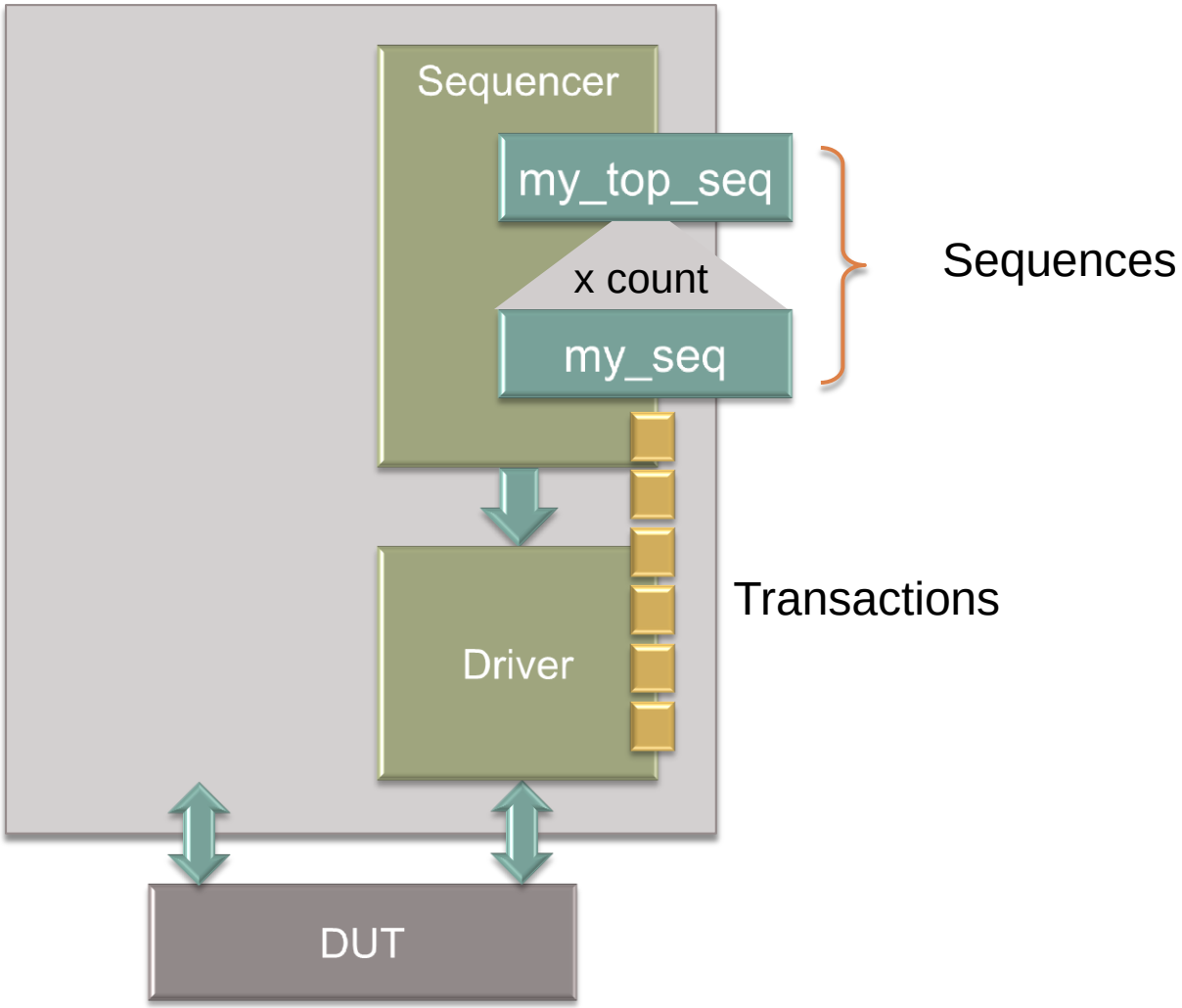


Checking and Coverage

Constrained random stimulus generation

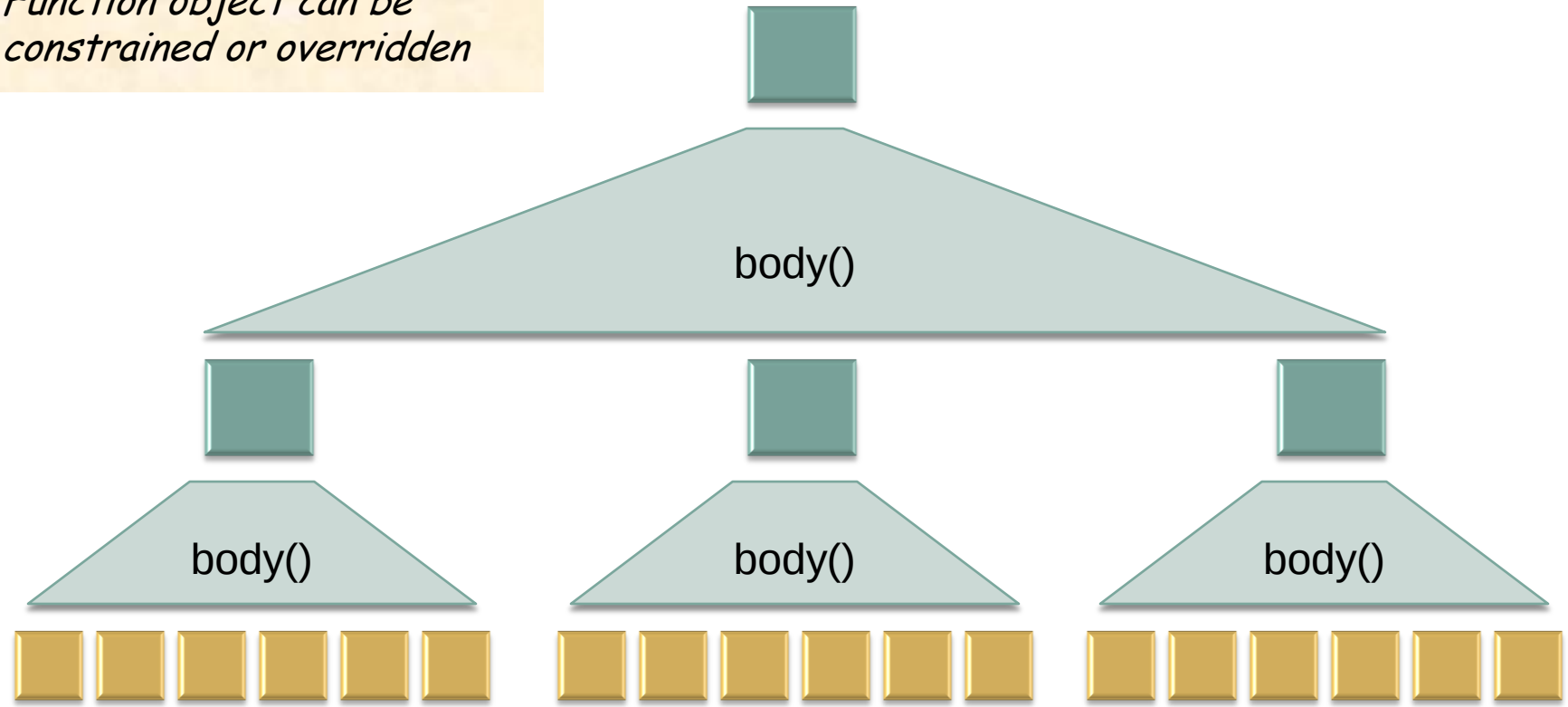
Agent-per-interface

Sequence versus Sequencer



Sequences

Function object can be constrained or overridden



Sequence of transactions

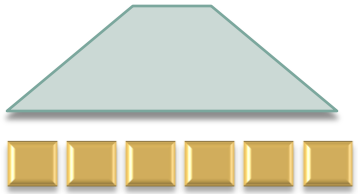
Transaction vs Sequence

Pattern 2a

```
class my_tx extends uvm_sequence_item;
  `uvm_object_utils(my_tx)

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

Pattern 2b

```
class my_seq extends uvm_sequence #(my_tx);
  `uvm_object_utils(my_seq)

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

  task body;
    ...
  endtask
  ...
endclass
```



Body of a Sequence

Pattern 2b

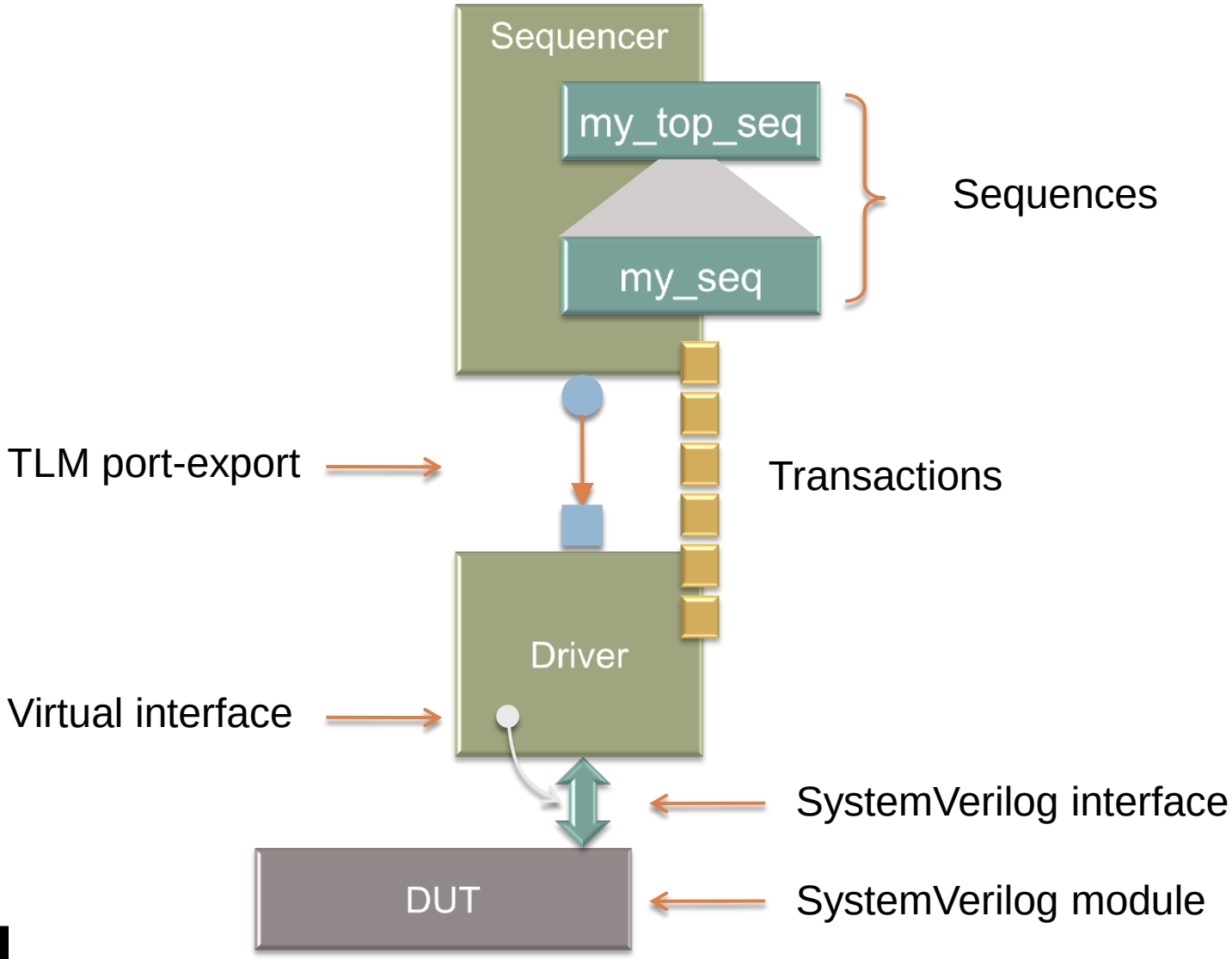
```
class my_seq extends uvm_sequence #(my_tx);
  `uvm_object_utils(my_seq)

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

  task body;
    repeat (6)
    begin
      req = my_tx::type_id::create("req");

      start_item(req);

      assert( req.randomize() with { data > 127; } );

      finish_item(req);
    end
  endtask

endclass
```

Factory-made transaction

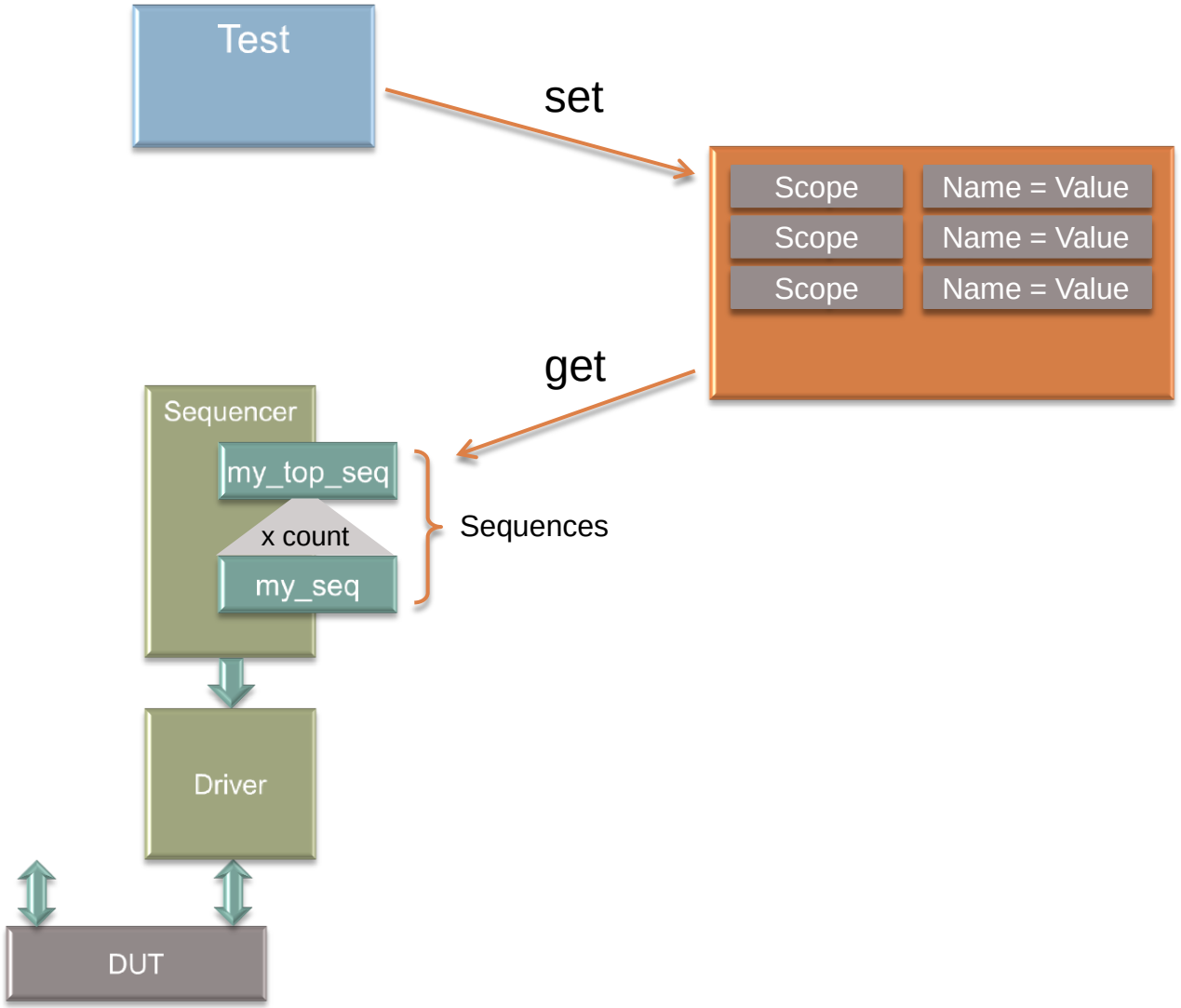
Handshake with driver

Late randomization

Sequencer – Driver – DUT



The Configuration Database



Driver 1

Pattern 1

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

  virtual  dut_if  dut_vi;
  ...
  function  void  build_phase( uvm_phase phase );
    super.build_phase(phase);
    uvm_config_db #(virtual dut_if)::get( this, "", "dut_vi", dut_vi );
  endfunction
  ...
```

Driver-DUT comms via interface

Interface set using config_db

Driver 2

```
task run_phase( uvm_phase phase );
  forever
  begin
    seq_item_port.get_next_item(req);

    @(posedge dut_vi.clock);
    dut_vi.cmd  = req.cmd;
    dut_vi.addr = req.addr;
    dut_vi.data = req.data;

    seq_item_port.item_done();
  end
endtask
endclass
```

Sequencer-driver comms via implicit port

Wiggle DUT pins through interface

or item_done(rsp);

Configuration Object

```
class  my_top_seq_config  extends  uvm_object;

  rand  int  count;
  rand …

endclass
```

A set of parameters / generics

Populate Configuration from a Test

Pattern 1

```
class  my_test  extends  uvm_test;
  `uvm_component_utils(my_test)
  ...
  function void build_phase (uvm_phase phase);
    super.build_phase(phase);
    begin
      my_top_seq_config  config = new;
      config.count = 6;

      uvm_config_db #(my_top_seq_config)::set(this,  "*.*seq*",  "config",  config);


      env = top::type_id::create("env", this);
    end
```

Could randomize the configuration

cf. parameter overrides / generic map

Retrieve Configuration Information

Pattern 2b

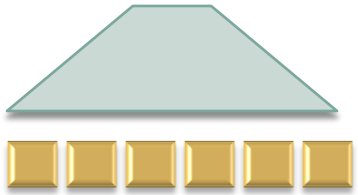
```
class  my_top_seq  extends  uvm_sequence #(my_top_tx);
  `uvm_object_utils(my_top_seq)
  `uvm_declare_p_sequencer(my_top_seqr)

  rand  int  count;
  constraint  how_many  { count inside { [4:10] }; }
  ...
  task body;
    my_top_seq_config  config;
    if ( uvm_config_db #(my_top_seq_config)::get(p_sequencer, "", "config", config) )
      count = config.count;
    …
  endtask
```

Factory Overrides

DOULOS

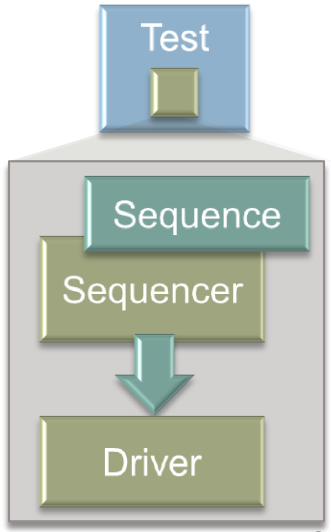
Pattern 2b

```
class my_seq extends uvm_sequence #(my_tx);
  `uvm_object_utils(my_seq)

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

  task body;
    req = my_tx::type_id::create("req");
    ...
  endtask
endclass
```



Pattern 1

```
class my_test extends uvm_test;
  ...
  function void build_phase (uvm_phase phase);
    ...
    my_tx::type_id::set_type_override( alt_tx::get_type() );

    my_component::type_id::set_inst_override(
                        alt_component::get_type(), "subsys.*", this );
...
```



Start Sequence from Test

Pattern 1

```
class  my_test  extends  uvm_test;
  `uvm_component_utils(my_test)
  ...
  my_env  env;
  ...
  task run_phase( uvm_phase phase );
    my_top_seq  seq;
    seq = my_top_seq::type_id::create("seq");

    assert( seq.randomize() with { count < 8; } );

    seq.start( env.sequencer );
  endtask
  ...
```

Randomize with constraint

Start sequence on sequencer

Running the Test

```
module top;
  import  uvm_pkg::*;
  import  my_pkg::*;

  dut_if  dut_if1 ();
  dut     dut1 ( ._if(dut_if1) );

  initial
  begin
    uvm_config_db #(virtual dut_if)::set(null, "*", "dut_vi", dut_if1);

    run_test();
  end
endmodule
```

UVM package

SV interface

Design-Under-Test

End-of-Test

Test ends when all objections dropped

```
task  run_phase(uvm_phase phase);
   @(posedge dut_vi.clock);
   forever
   begin
      seq_item_port.get_next_item(req);
      phase.raise_objection(this);
      ...
      @(posedge dut_vi.clock);
      dut_vi.data = req.data;
      phase.drop_objection(this);
      ...
   end
endtask
```

```
task body;
   uvm_test_done.raise_objection(this);
   repeat(n)
   begin
      req = my_top_req::type_id::create("req");
      start_item(req);
      assert( req.randomize() );
      finish_item(req);
   end
   uvm_test_done.drop_objection(this);
endtask
```

Summary of Coding Idioms

Pattern 1

```
class my_comp extends uvm_component;
  `uvm_component_utils(my_comp)

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

  function void build_phase(...);
    ...
endclass
```

Pattern 2a

```
class my_tx extends uvm_sequence_item;
  `uvm_object_utils(my_tx)

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

Pattern 2b

```
class my_seq extends uvm_sequence #(my_tx);
  `uvm_object_utils(my_seq)

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

Other Features of UVM

- Message reporting
- Further sequence mechanisms (lock, arbitration, sequence library)
- User-defined phasing
- Further transaction-level communication, including TLM-2.0
- Printing, recording, and packing/unpacking of transactions
- Synchronization mechanisms (events, barriers, heartbeats)
- Customized reporting and report catching
- Command line processor
- The register abstraction layer

What Next?

- Download UVM

  http://www.accellera.org/downloads/standards/uvm/

- Download *Easier UVM* examples

  http://www.doulos.com/knowhow/sysverilog/uvm/easier_uvm

- Future webinars from Doulos

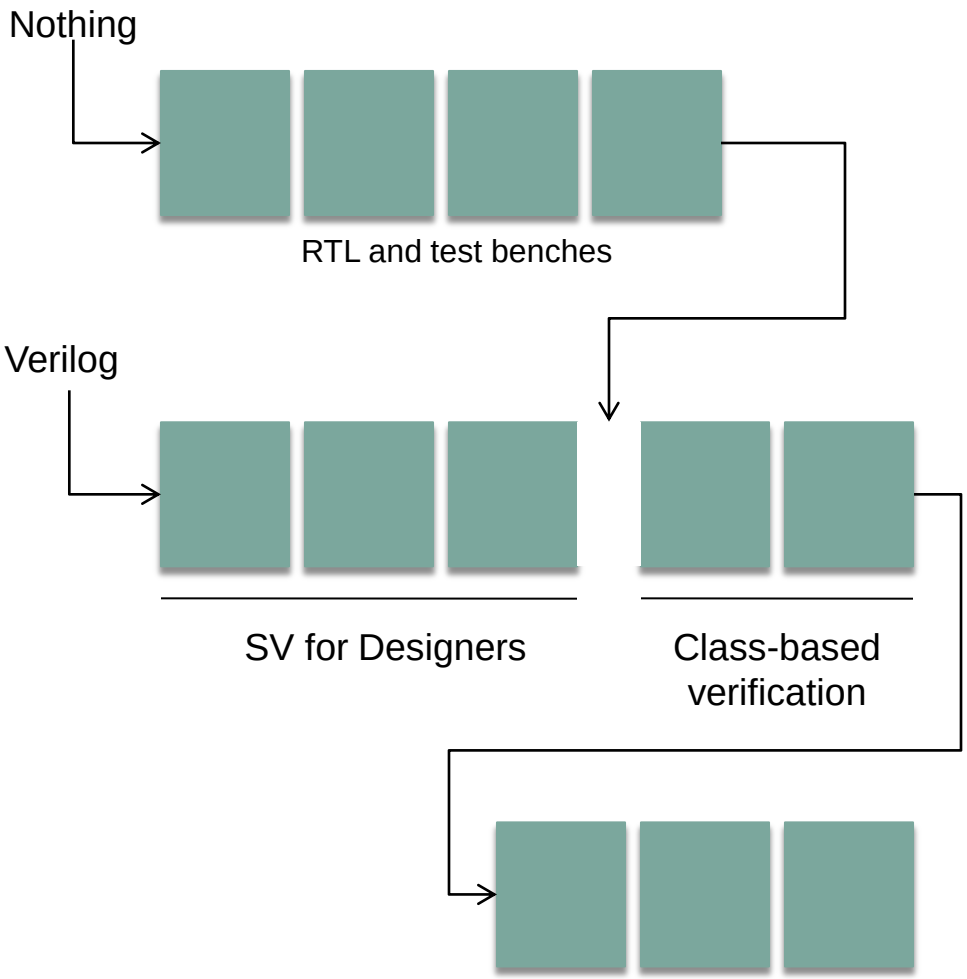
SystemVerilog Training Portfolio

Public Classes

SystemVerilog for FPGA/ASIC Design

Comprehensive SystemVerilog

UVM Adopter Class



Hardware Design

» VHDL » Verilog » SystemVerilog
» Altera » Microsemi » Xilinx

Embedded Systems and ARM

» C » C++ » UML » RTOS » Linux
» ARM Cortex A/R/M series

ESL & Verification

» SystemC » TLM-2.0 » SystemVerilog
» OVM/VMM/UVM » Perl » Tcl/Tk