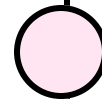
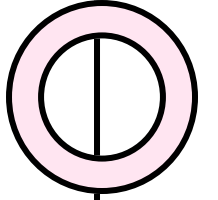
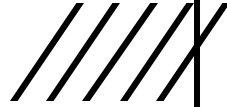


UVM LIBRARY BASICS

(UNIVERSAL VERIFICATION
METHODOLOGY)

-DIVYANSH SINGHAL
(IMT2021522)

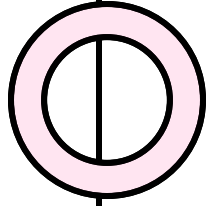
-CHINMAY SULTANIA
(IMT2021540)



○ What is UVM?

- The Universal Verification Methodology (UVM) is a complete methodology that codifies the best practices for efficient and exhaustive verification.
- One of the key principles of UVM is to develop and leverage reusable verification components-also called UVM Verification Components (UVCs).
- The UVM is targeted to verify small designs and large-gate-count, IP-based system-on-chip (SoC) designs.





Key features of UVM

1. Data Design
2. Stimulus Generation
3. Building and running the testbench
4. Coverage model design and Checking strategies
5. User Example
6. Open, compatible and Portable

Goal Of UVM: Automation



Coverage Driven
verification (CDV)
environments



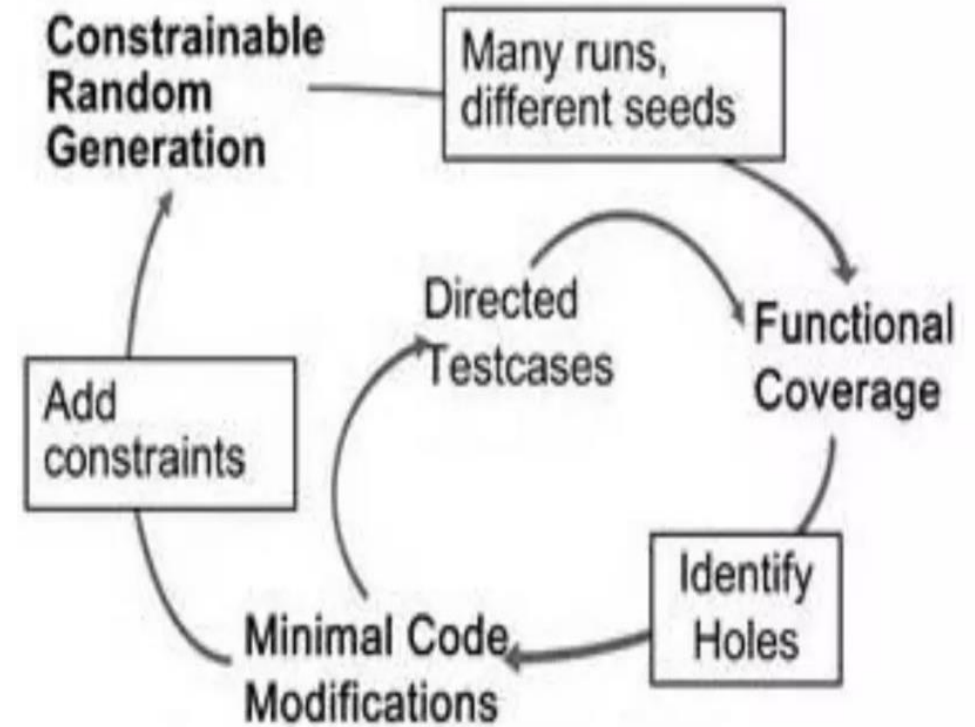
Automated
stimulated
Generation



Independent
checking

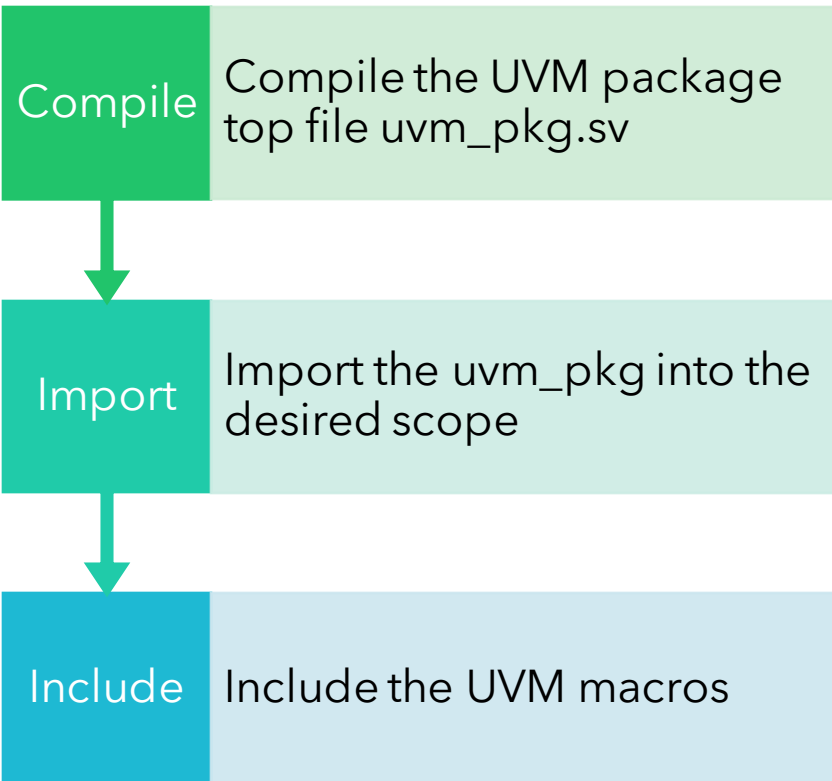


Coverage Collection






Using the UVM Library



hello_world.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
```



○ Guidelines for Using the UVM Library

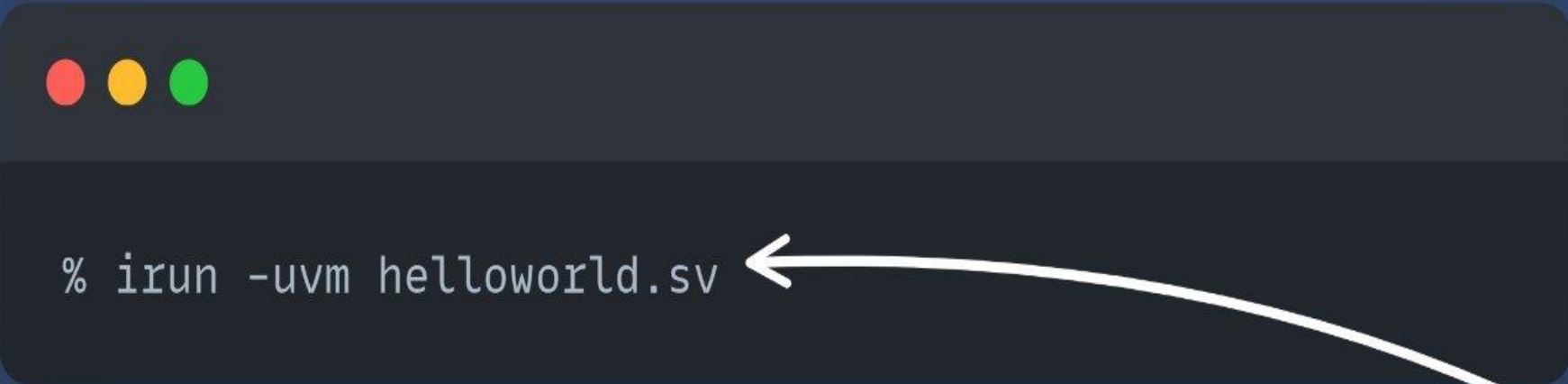
- To prevent name collisions, avoid importing the uvm_pkg into the global scope. This is true for any package being imported.
- The top UVM files are typically enclosed by the following:

```
`ifndef <FILE NAME> SVH  
`define <FILE NAME> SVH  
....body of the file  
`endif
```

- This allows including the UVM library from multiple locations and avoids multiple declarations by compiling the files only once. We recommend using this technique in the user's UVC files.



- To run this test on the Cadence Incisive® Enterprise Simulator (IES) with the UVM package that is delivered with IES:

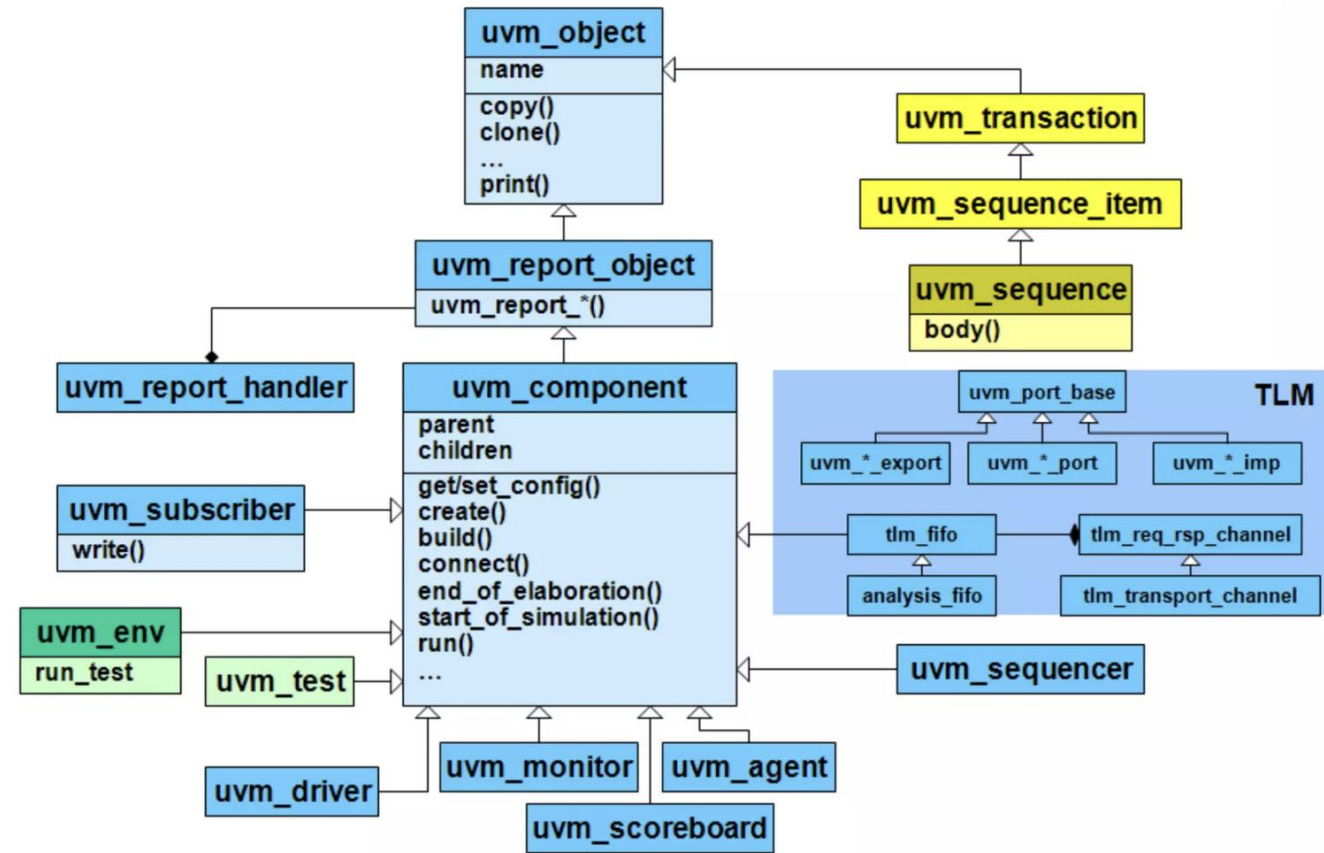


```
% irun -uvm helloworld.sv
```

A terminal window with a dark background and three colored window control buttons (red, yellow, green) in the top-left corner. The command `% irun -uvm helloworld.sv` is entered in a monospaced font.

The command "irun" is used to compile and simulate SystemVerilog code.

UVM LIBRARY BASE CLASSES



Note: We do not show the **uvm_transaction** class in this diagram. While this class is still part of the library, it was voted to be deprecated by the **Accellera TSC**.

The uvm_object class

```
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;

    // 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

endclass : apb_transfer
```

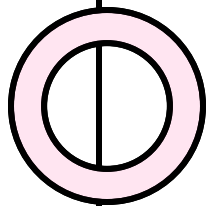
```
typedef enum bit {APB_READ, APB_WRITE} apb_direction_enum;
class apb_transfer;
    rand bit [31:0]      addr;
    rand bit [31:0]      data;
    rand apb_direction_enum  direction;
    // Control field - does not translate into signal data
    rand int unsigned      transmit_delay;

    function void print();
        $display("%s transfer: addr=%h data=%h", direction.name(),
    endfunction : print

endclass : apb_transfer
```

Non-UVM Class Definition

APB Transfer derived from uvm_object



Uvm_object definition guidelines

Use UVM_DEFAULT as the flag argument (instead of UVM_ALL_ON) for all ``uvm_field_*` macros. This allows the UVM architects to add automation that by default might not be enabled by default. Other flags are used to remove undesired automation.

Set the class name as the default value in the constructor argument.

Do not forget to call `super.new (name)` as the first statement in every constructor declaration.

There is a different macro for classes that have type parameters ``uvm_object_param_utils*`

If you are using a reference to another object, make sure to use the UVM_REFERENCE flag so that deep operations are not done.



UVM Field Automation

The ``uvm_object_utils_begin(type)` and ``uvm_object_utils_end` macros are used to declare common operations declared for UVM objects.

Implements `get_type_name()` which returns the object type as a string

Implements `create()` which allocates an object of the specified type by calling its constructor

Registers the type with the factory so it can be overridden elsewhere in the testbench

Implements a static `get_type()` method needed for the factory operation

Ran the given code on EDA playground using

- Aldec Riviera Pro 2023.04 simulator

```
typedef enmodule automation_example;
import uvm_pkg::*;
`include "uvm_macros.svh"
import apb_pkg::*;

apb_transfer my_xfer, tx1, tx2, tx3;

initial begin
    my_xfer = apb_transfer::type_id::create("my_xfer");
    if (!my_xfer.randomize())
        `uvm_fatal("RANDFAIL", "can not randomize my_xfer")
    tx1 = my_xfer; // tx1 and my_xfer share the same memory
    tx2 = apb_transfer::type_id::create("tx2");
    tx2.copy(tx1); // copies fields from tx1 to tx2
    $cast(tx3, tx1.clone()); // creates a new apb_transfer and copy all
                             // fields from tx1 to tx3

    if (!tx3.compare(tx2))
        `uvm_error("CompareFailed", "The comparison failed")
    my_xfer.print(); // Prints my_xfer in a table format
    my_xfer.print(uvm_default_tree_printer); // Prints in "tree" format
    my_xfer.print(uvm_default_line_printer); // Prints in "line" format
end

endmodule : automation_example
```

```
# KERNEL: ASDB file was created in location /home/runner/dataset.asdb
```

```
# KERNEL: -----
```

# KERNEL: Name	Type	Size	Value
----------------	------	------	-------

```
# KERNEL: -----
```

# KERNEL: my_xfer	apb_transfer	-	@335
-------------------	--------------	---	------

# KERNEL: addr	integral	32	'hb84bc20c
----------------	----------	----	------------

# KERNEL: data	integral	32	'hb482eafc
----------------	----------	----	------------

# KERNEL: direction	apb_direction_enum	1	APB_READ
---------------------	--------------------	---	----------

# KERNEL: transmit_delay	integral	32	'h87f3f9e8
--------------------------	----------	----	------------

```
# KERNEL: -----
```

```
# KERNEL: my_xfer: (apb_transfer@335) {
```

```
# KERNEL:   addr: 'hb84bc20c
```

```
# KERNEL:   data: 'hb482eafc
```

```
# KERNEL:   direction: APB_READ
```

```
# KERNEL:   transmit_delay: 'h87f3f9e8
```

```
# KERNEL: }
```

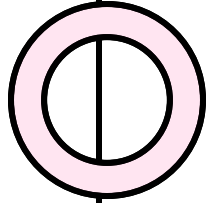
```
# KERNEL: my_xfer: (apb_transfer@335) { addr: 'hb84bc20c data: 'hb482eafc direction: APB_READ transmit_delay: 'h87f3f9e8
```

```
# KERNEL: Simulation has finished. There are no more test vectors to simulate.
```

```
# VSIM: Simulation has finished.
```

```
Done
```





Advantages of using UVM Field Automation

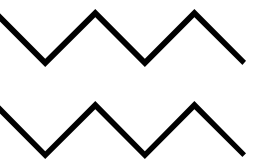
Productivity

Extensibility

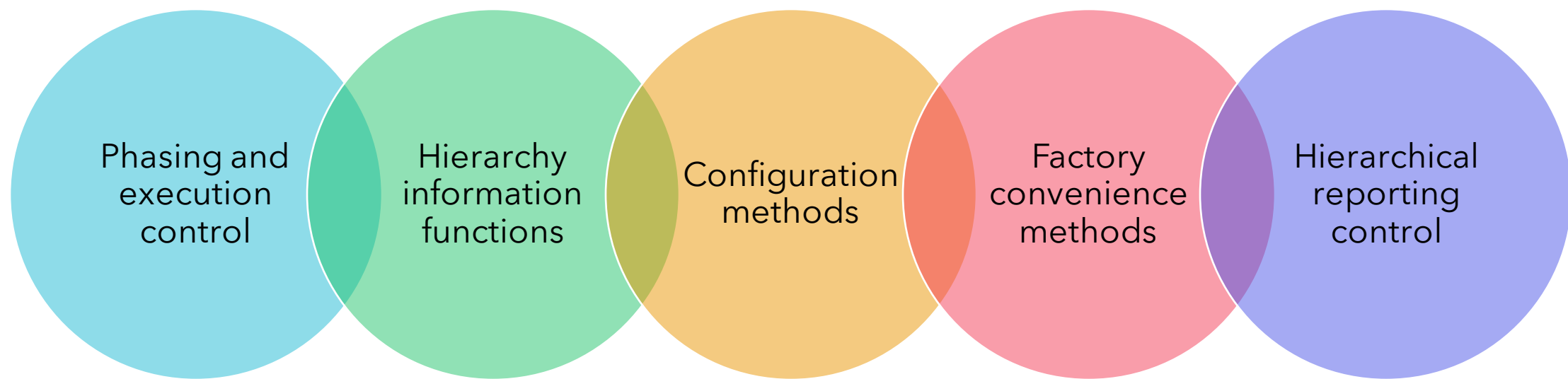
Implementation

Maintainability

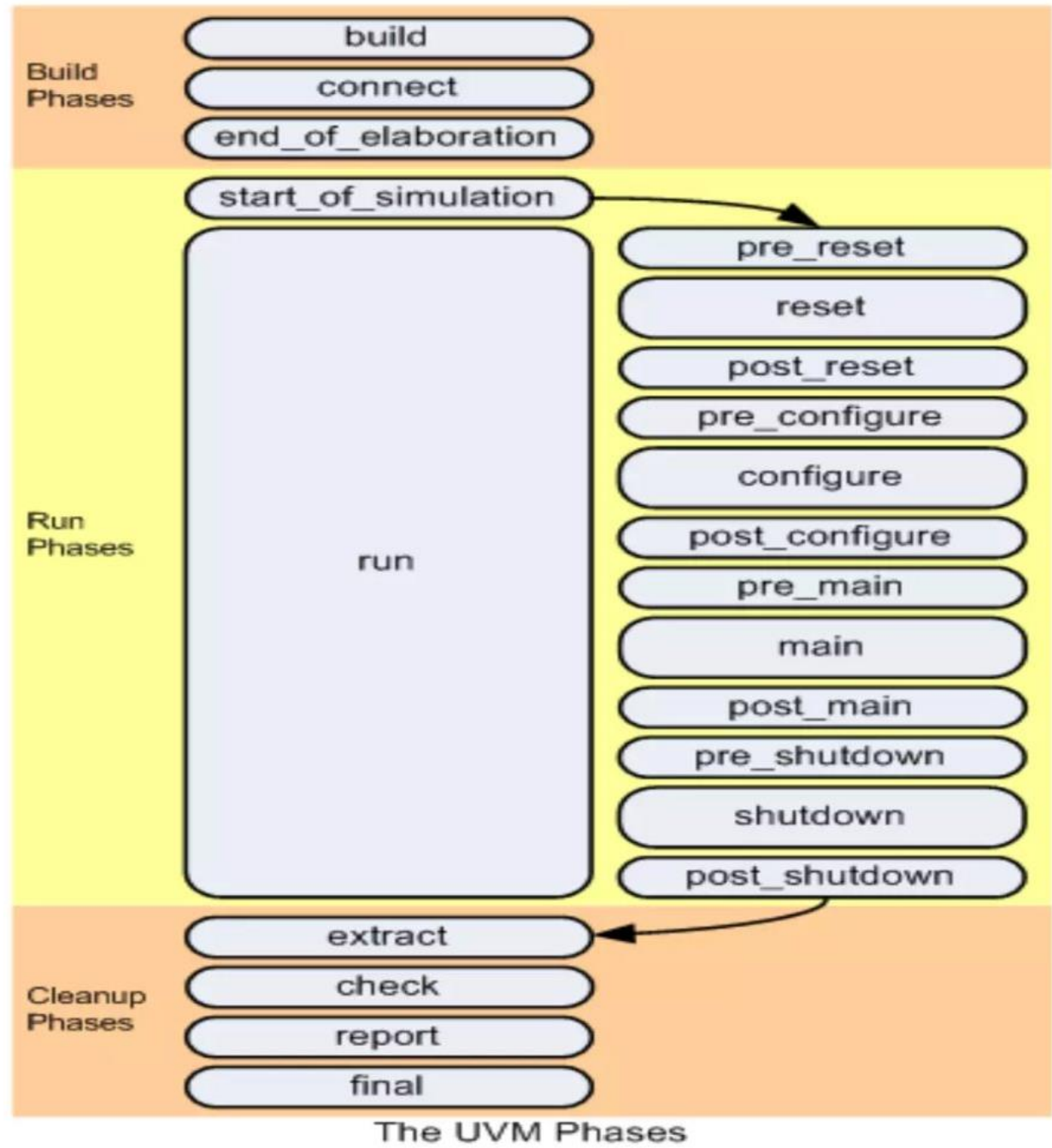
Code Correctness



The uvm_component class

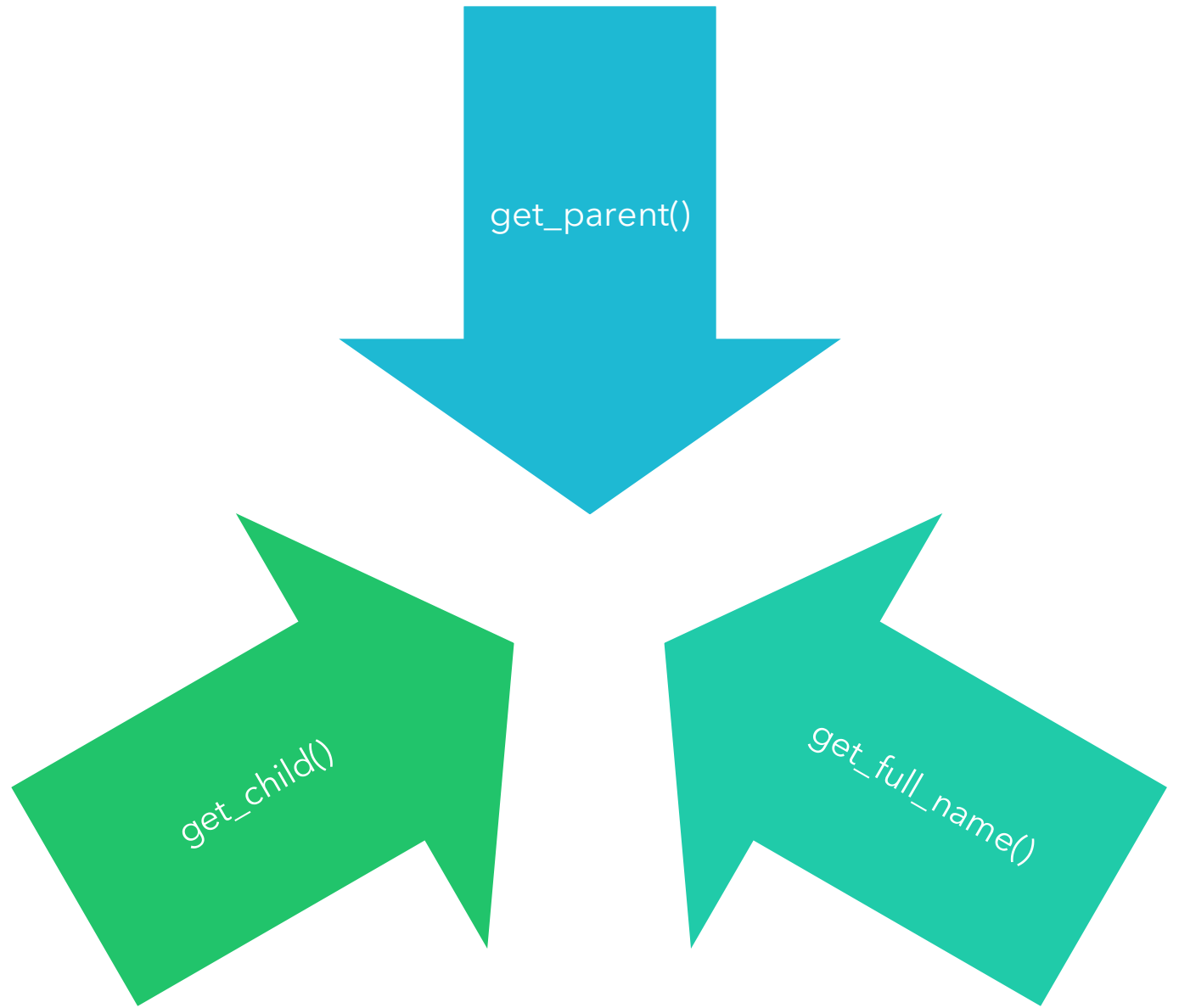
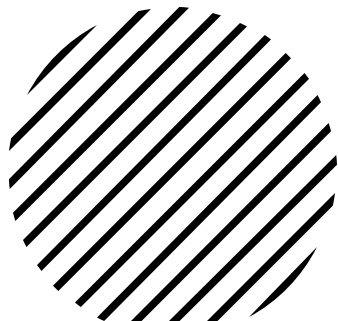


SIMULATION PHASE METHODS





Hierarchy Information functions





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

class master_comp extends uvm_component;
    `uvm_component_utils(master_comp)
    function new(string name, uvm_component parent);
        super.new(name, parent);
    endfunction : new

    // UVM run_phase() method
    task run_phase (uvm_phase phase);
        `uvm_info("MASTER", "run_phase: Executing.", UVM_LOW)
    endtask : run_phase
endclass : master_comp

class slave_comp extends uvm_component;

    `uvm_component_utils(slave_comp)
    function new(string name, uvm_component parent);
        super.new(name, parent);
    endfunction : new

    // UVM run_phase() method
    task run_phase (uvm_phase phase);
        `uvm_info("SLAVE", "run_phase: Executing.", UVM_LOW)
    endtask : run_phase
endclass : slave_comp

class simple_if_comp extends uvm_component;

    master_comp master;
    slave_comp slave;

    `uvm_component_utils(simple_if_comp)

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

```
function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    my_uvc = simple_if_comp::type_id::create("my_uvc", this);
endfunction : build_phase

function void end_of_elaboration_phase(uvm_phase phase);
    super.end_of_elaboration_phase(phase);
    `uvm_info("TBENCH", {"end_of_elaboration_phase: Hierarchy\n",
        this.sprint()}, UVM_LOW)
endfunction : end_of_elaboration_phase

// UVM run_phase() method
task run_phase (uvm_phase phase);
    `uvm_info("TBENCH", "run_phase: Executing.", UVM_LOW)
endtask : run_phase

endclass : testbench_comp
endpackage : my_pkg
```





UVM

TOP

TB

TEST

ENV

PACKET

SEQUENCES

SCOREBOARD

PASS FAIL

MONITOR

AGENT

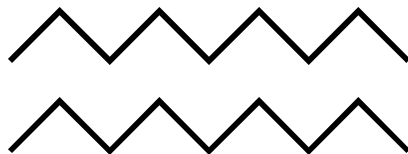
SEQUENCER

DRIVER

DUT

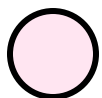
TLM connection





EXAMPLE CODE FOR UVM

TESTBENCH ARCHITECTURE



```
1 module verilog_testbench();
2
3 // Inputs
4 reg a;
5 reg b;
6 reg c;
7 reg d;
8 reg clock;
9
10 // Outputs
11 wire out;
12
13 // Instantiate the Unit Under Test (UUT)
14 verilog_module uut( .a(a), .b(b), .c(c), .d(d), .out(out), .clock(clock));
15
16 always #1 clock = ~clock;
17
18 initial begin
19
20     clock = 1'b0;
21     #10;
22     a = 1'b1;
23     b = 1'b0;
24     c = 1'b1;
25     d = 1'b0;
26     #10;
27
28     repeat(100)
29     begin
30         a = $random;
31         b = $random;
32         c = $random;
33         d = $random;
34         #5;
35     end
36
37     #30;
38     $finish;
39
40 end
41 endmodule
```

TOP

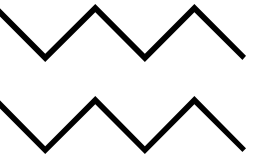
DUT

TEST

DRIVER

SEQUENCE

PACKET



Transaction level Modeling in UVM

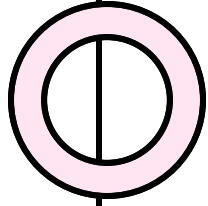
In TLM, transactions in the system are modeled using method calls and class objects. There are several significant benefits that come from modeling at the transaction level :

TLM models are more concise and simulate faster than RTL models.

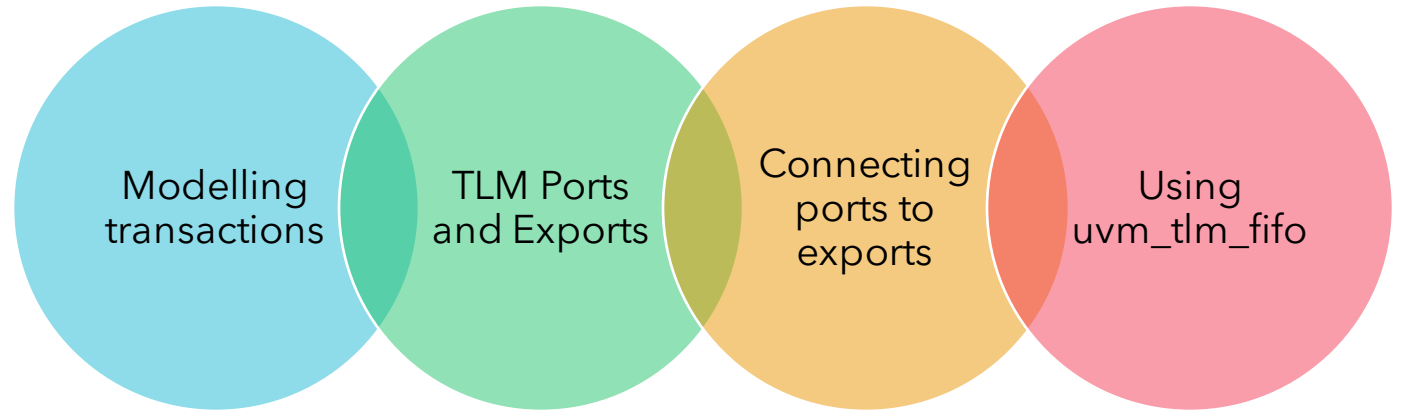
TLM models are at a higher level of abstraction. This makes it easier to write the models and easier for other engineers to understand them.

TLM models tend to be more reusable since unnecessary details which hinder reuse are moved outside of the models. Also, TLM enables use of OOPS such as inheritance and separation of interfaces from implementation.





Key TLM concepts in UVM

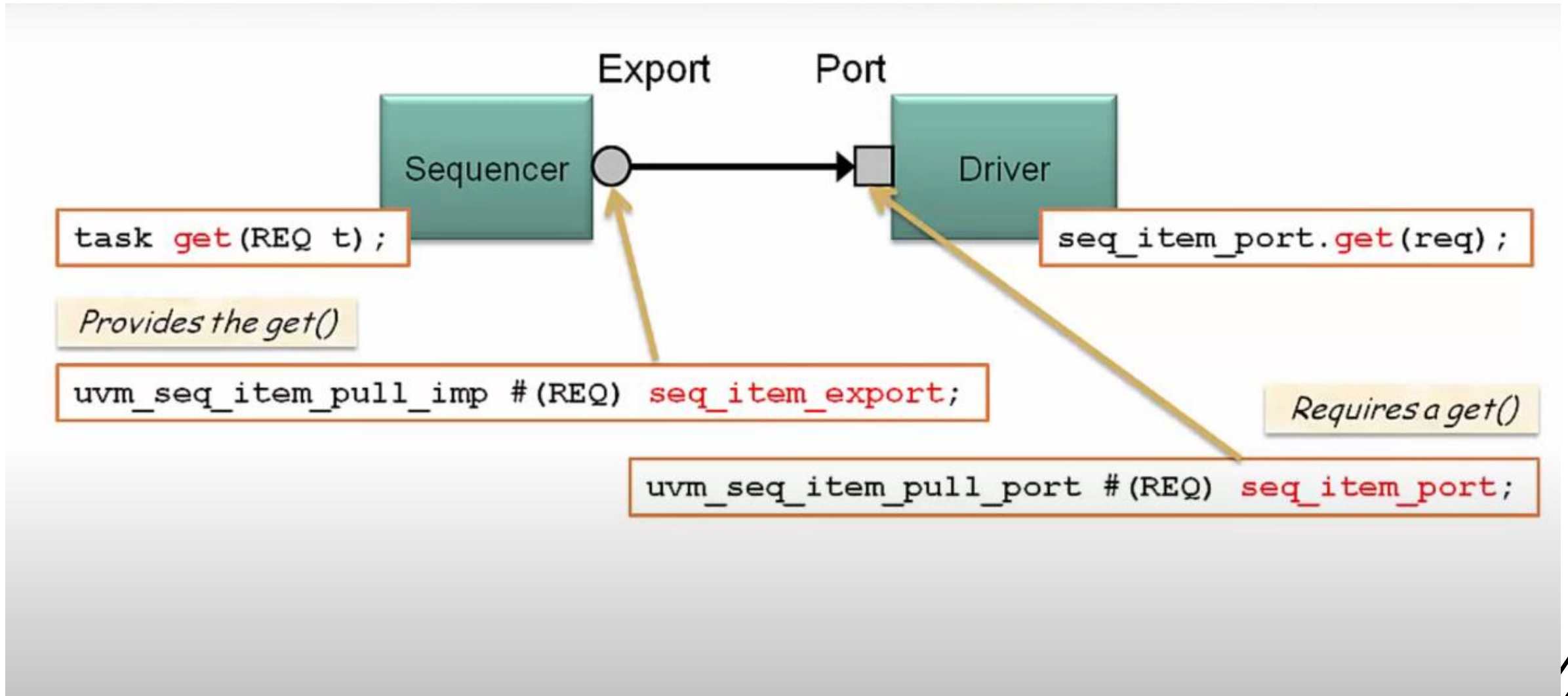


○ Modelling Transactions

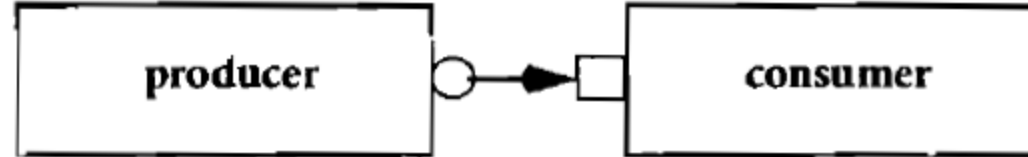
```
1  class simple_packet extends uvm_sequence_item;
2    rand int src_addr;
3    rand int dst_addr;
4    rand byte unsigned data[];
5    constraint addr_constraint { src_addr != dst_addr; }
6    ...
7  endclass
```



○ TLM Ports and Exports



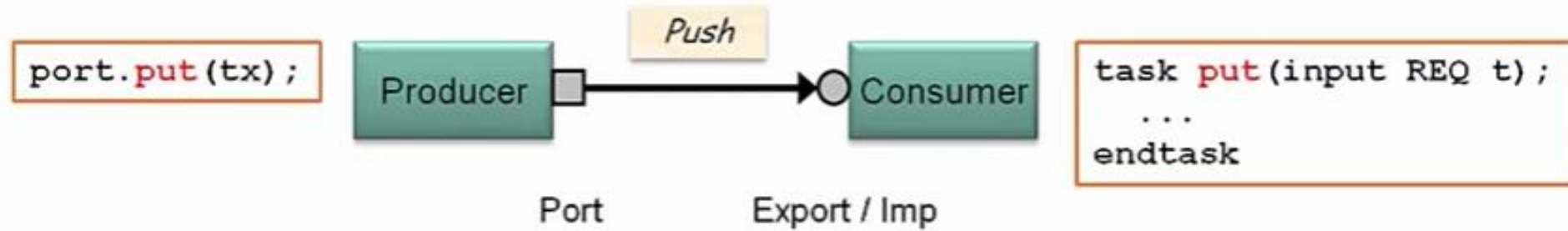
○ TLM Ports and Exports



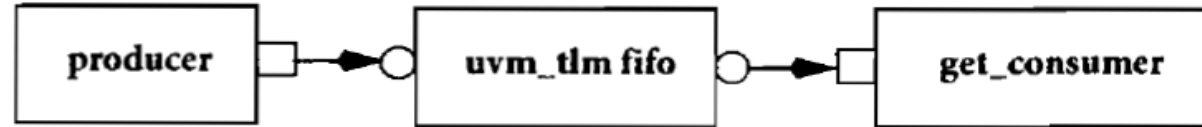
```
class parent_comp extends uvm_component;
  producer producer_inst;
  consumer consumer_inst;
  ...
  virtual function void connect();
    producer_inst.put_port.connect(consumer_inst.put_export);
  endfunction
endclass
```



○ TLM Ports and Exports



Using the uvm_tlm_fifo

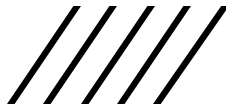


Example 4-10 uvm_tlm_fifo Usage

```
class producer_consumer_2 extends uvm_component;
  producer producer_inst;
  consumer_2 consumer2_inst;
  uvm_tlm_fifo #(simple_packet) fifo_inst;  // fifo stores simple_packets

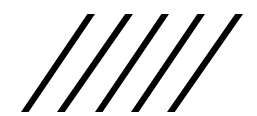
  function new(string name, uvm_component parent);
    producer_inst = new("producer_inst", this);
    consumer2_inst = new("consumer2_inst", this);
    fifo_inst = new("fifo_inst", this, 16);  // set fifo depth to 16
  endfunction

  virtual function void connect();
    producer_inst.put_port.connect(fifo_inst.put_export);
    consumer2_inst.get_port.connect(fifo_inst.get_export);
  endfunction
endclass
```





UVM Factory

- UVM factory is an advanced implementation of the classic software factory design pattern that is used to create generic code, deferring to run-time to decide the exact subtype of the object that will be allocated.
- 

○ UVM Non-factory allocation

```
1  class driver extends uvm_component
2      `uvm_component_utils(driver)
3      function new(string name, uvm_component parent);
4          super.new(name, parent);
5      endfunction
6      virtual task drive_transfer();
7          ...
8      endtask
9  endclass: driver

10 class agent extends uvm_component; // bad example that uses new()
11     `uvm_component_utils(agent)
12     driver my_driver;

13     function new(string name, uvm_component parent);
14         super.new(name, parent);
15         // create the driver
16         my_driver = new ("my_driver",this); // using new()
17     endfunction
18 endclass: agent
```



○UVM factory usage

To use the factory, these steps should be followed:

Register all classes within the factory by using the utility macros ``uvm_object_utils` and ``uvm_component_utils` for objects and components, respectively.

Create objects and components using the `create ()` API.

```
my_driver = driver::type_id::create("my_driver",  
this);
```

Use the type and instance override calls to override the base types with a derived types.



○UVM factory usage

```
1 class agent extends uvm_component;
2     `uvm_component_utils(agent)
3     driver my_driver;

4     function new(string name, uvm_component parent);
5         super.new(name, parent);
6 // create the driver
7         my_driver = driver::type_id::create("my_driver",this);
8     endfunction
9 endclass: agent
```





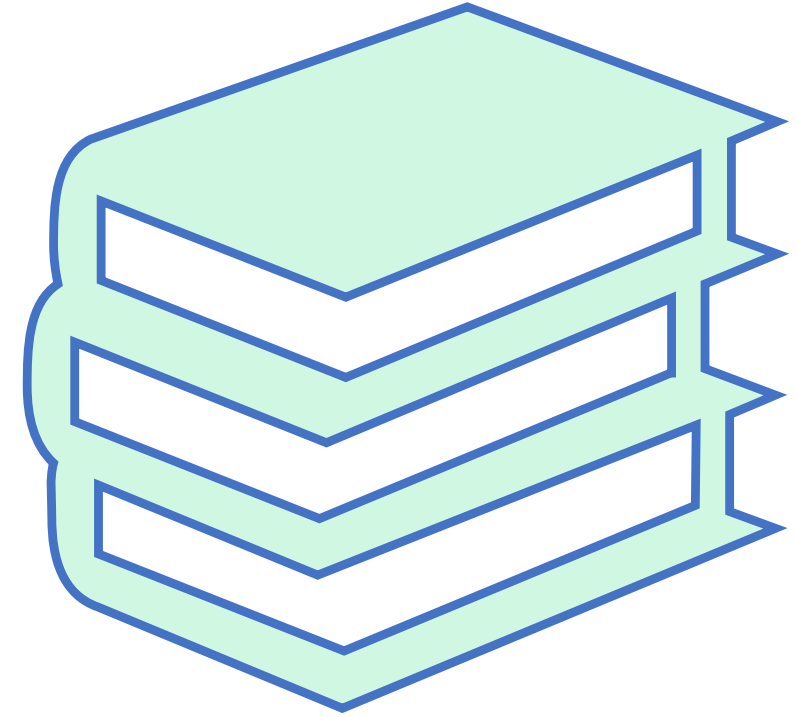
Summary

The chapter introduces key feature of the UVM library. While the UVM features are capable and flexible, they were created as enablers to the higher-level methodology that is needed to speed up verification for large and small designs.



References

1. <https://www.slideshare.net/arrowdevices/uvm-methodology-tutorial>
2. <https://www.chipverify.com/uvm/base-classes>
3. https://github.com/4get/uvm_book_examples/tree/master
4. <https://youtube.com/playlist?list=PLuYB6t6povcLgoHWLJgk-VeMQ0Rscjw03&si=5Axo-sX2zY47Q6Lc>
5. <https://www.edaplayground.com/s/example/546>
6. A Practical guide to Adopting the Universal Verification Methodology (UVM) - Sharon Rosenberg & Kathleen A Meade



**THANK
YOU**

