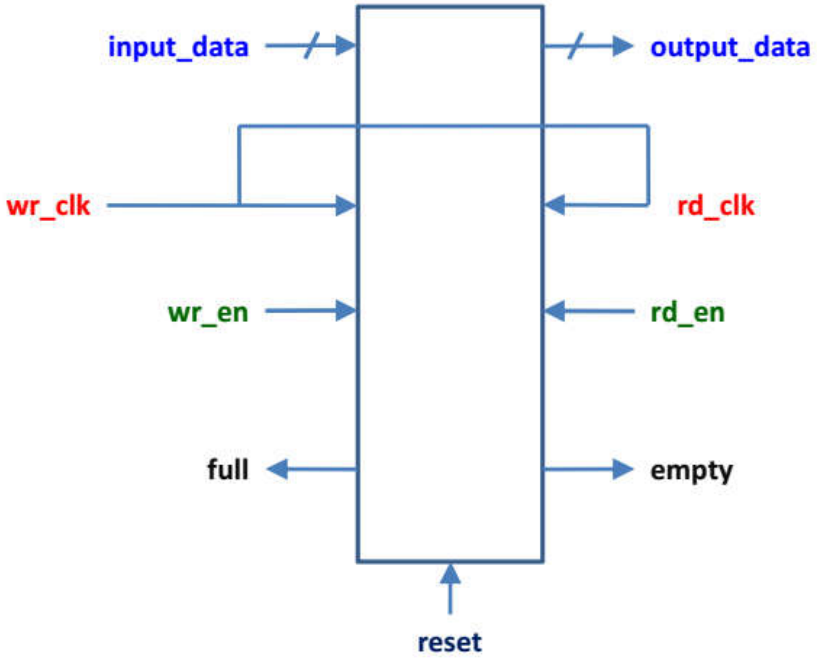
UVM (Universal Verification Methodology)

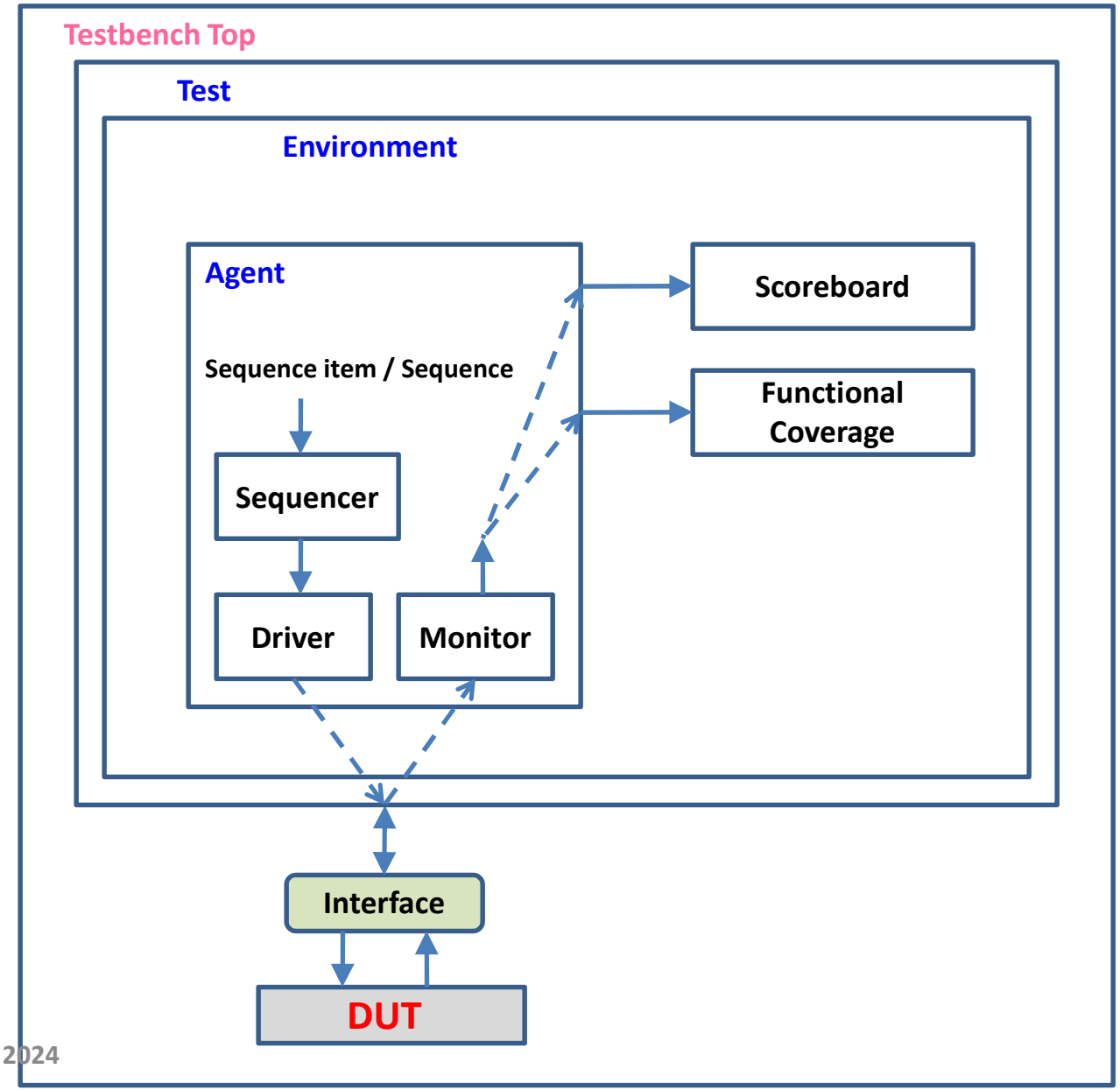
For whom can know how to program

Tuan Nguyen-viet

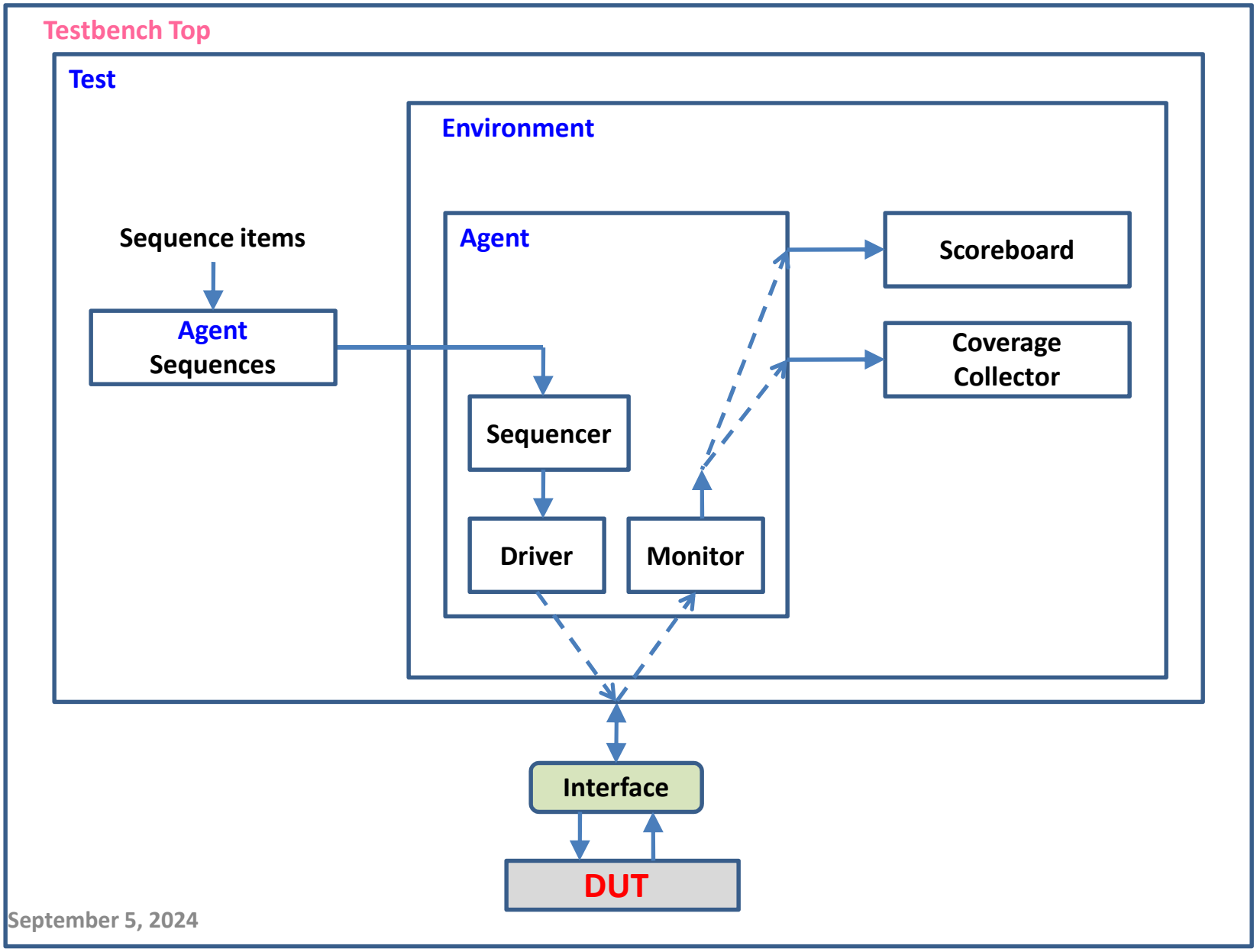
DUT – Sync FIFO



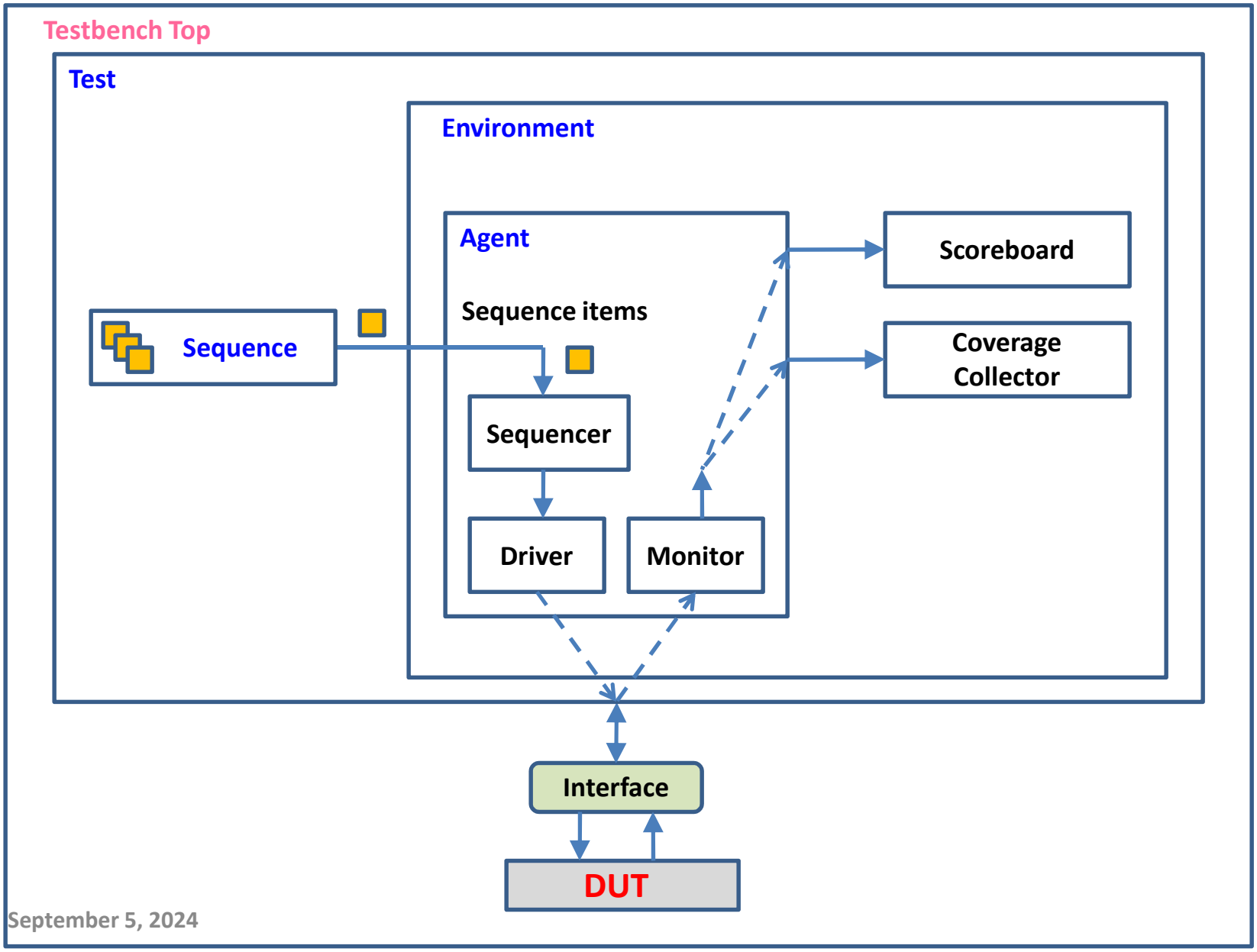
UVM - Simple Architecture w/ Single Agent



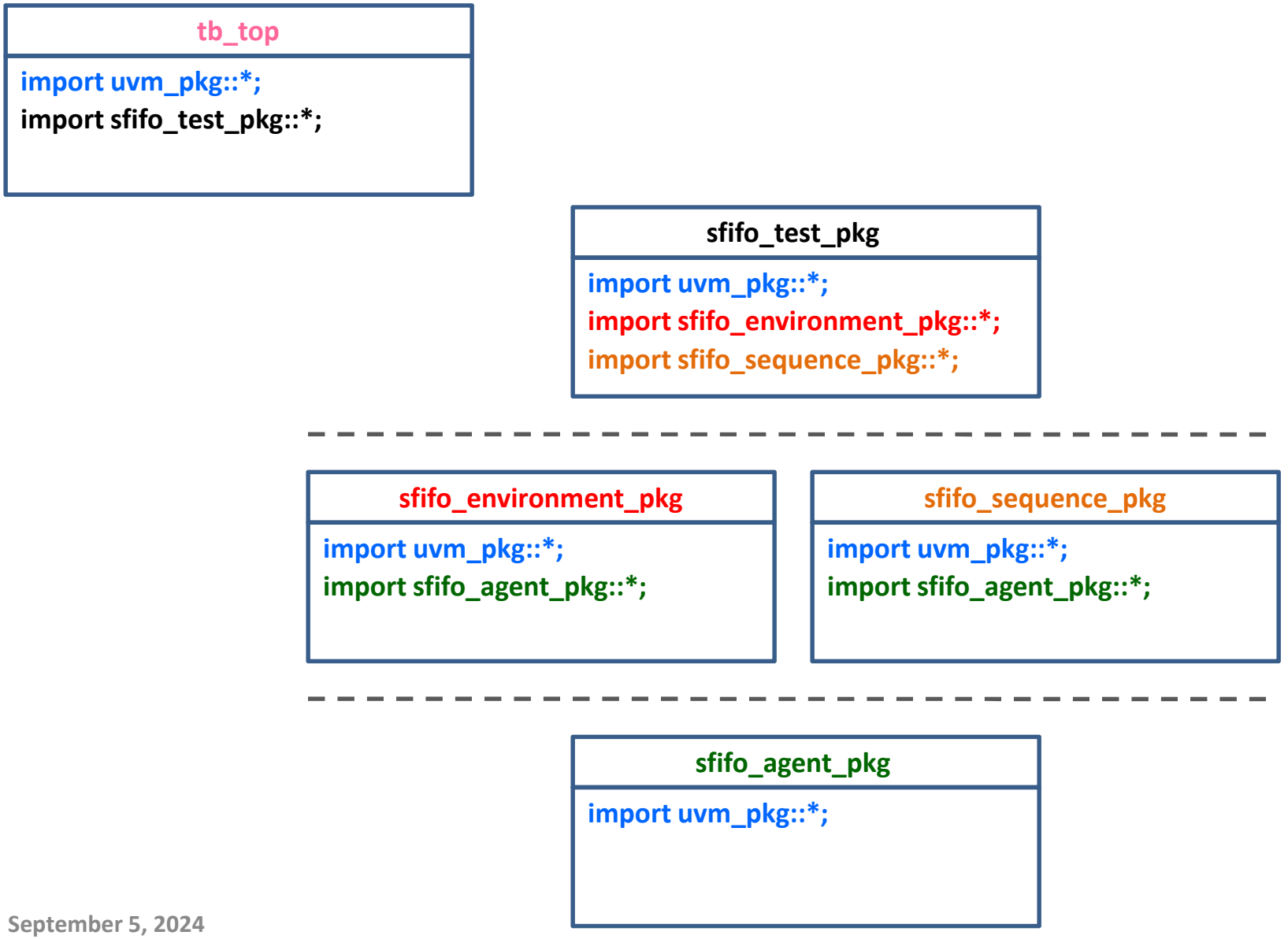
UVM - Simple Architecture w/ Single Agent (2)



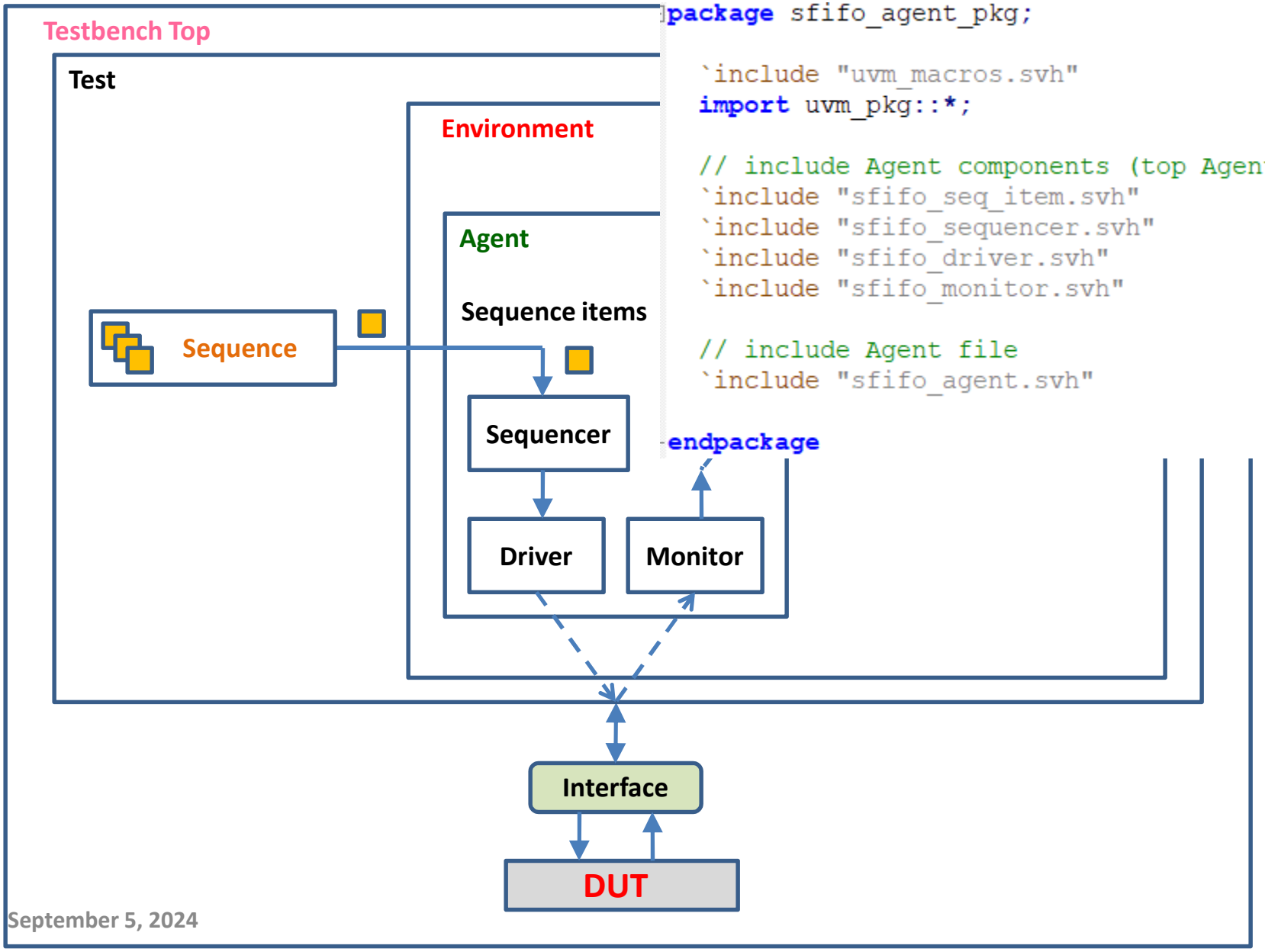
UVM - Simple Architecture w/ Single Agent (3)



Package Hierarchy

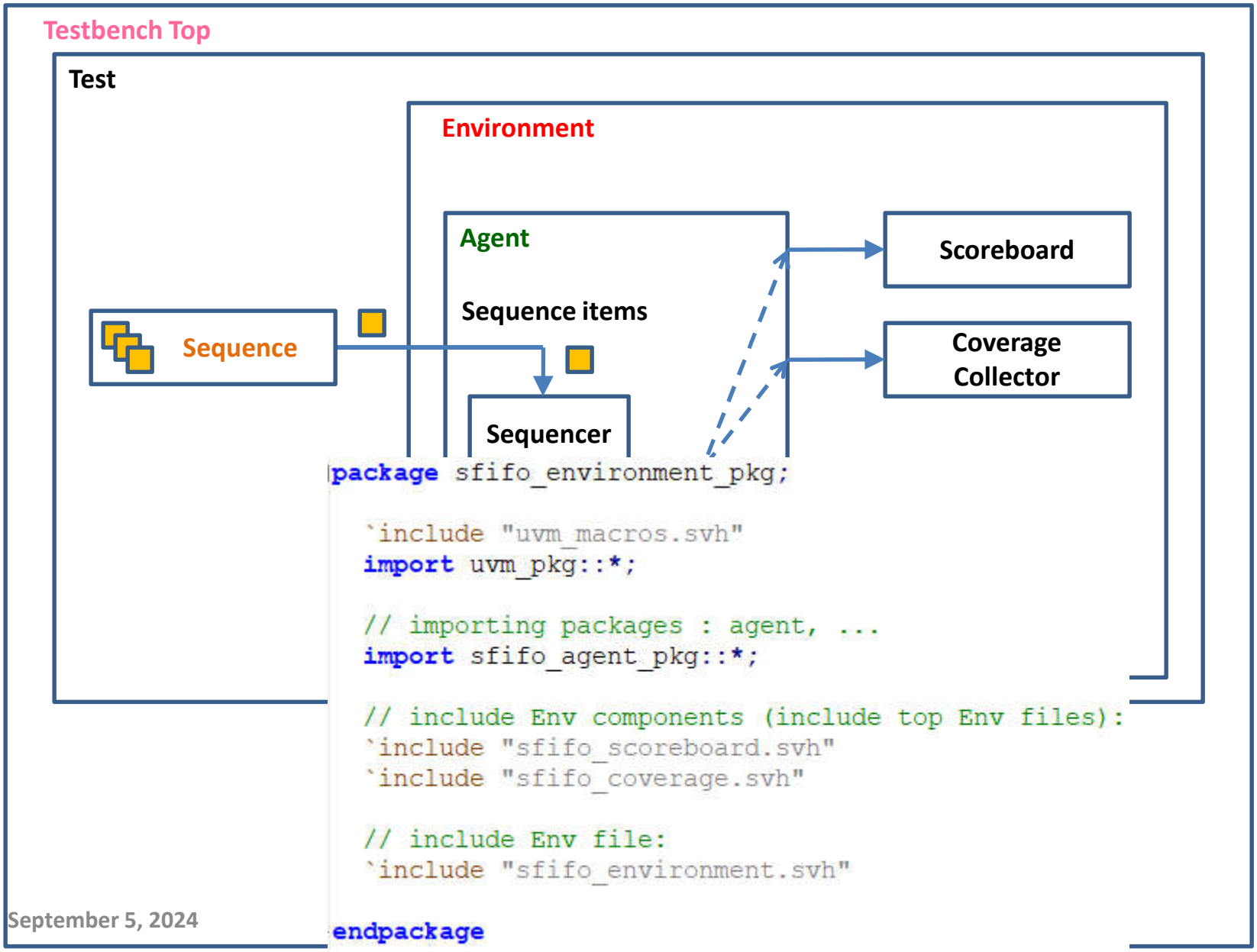


Agent Package



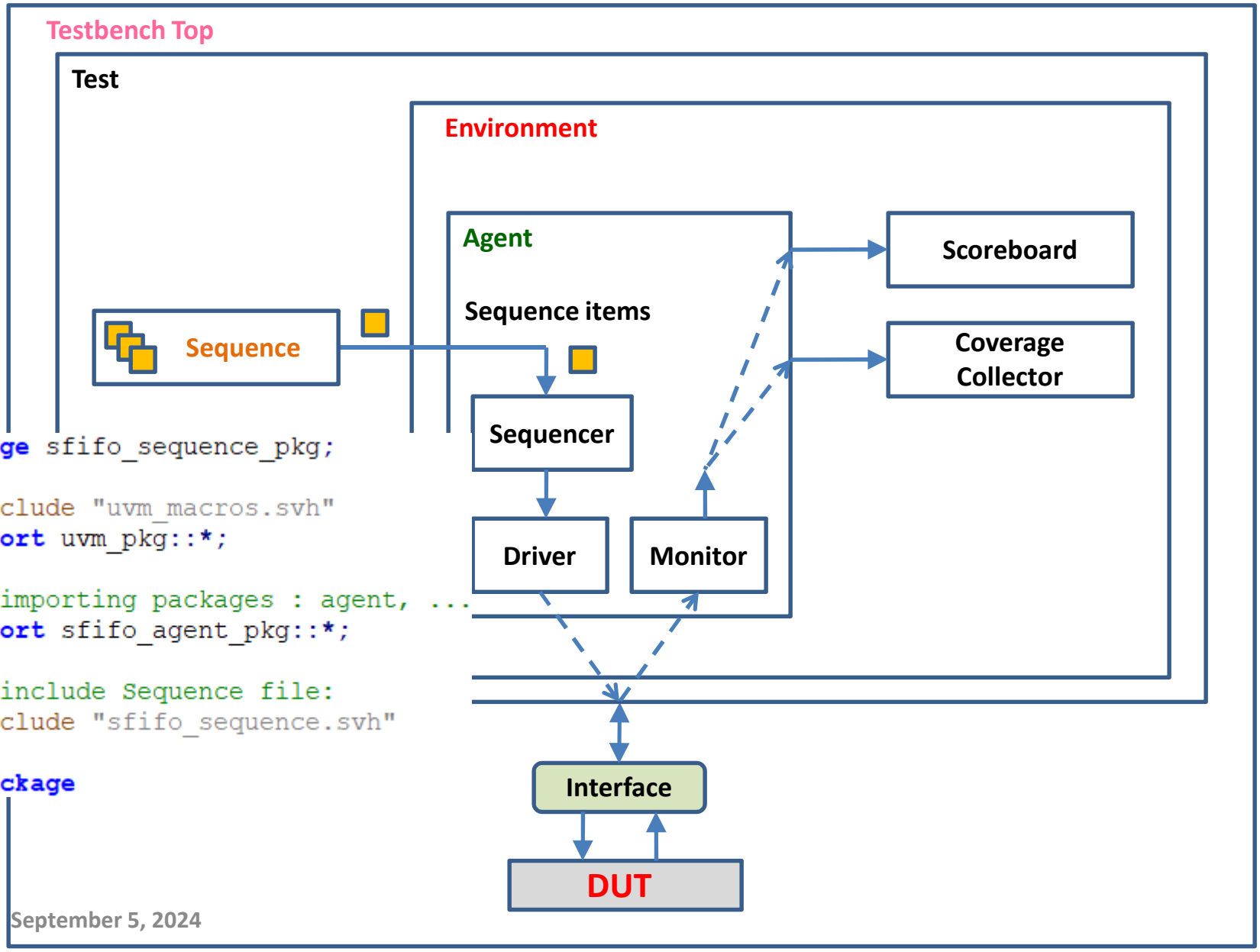

```
package sfifo_agent_pkg;

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

  // include Agent components (top Agent files)
  `include "sfifo_seq_item.svh"
  `include "sfifo_sequencer.svh"
  `include "sfifo_driver.svh"
  `include "sfifo_monitor.svh"

  // include Agent file
  `include "sfifo_agent.svh"

endpackage
```

Environment Package



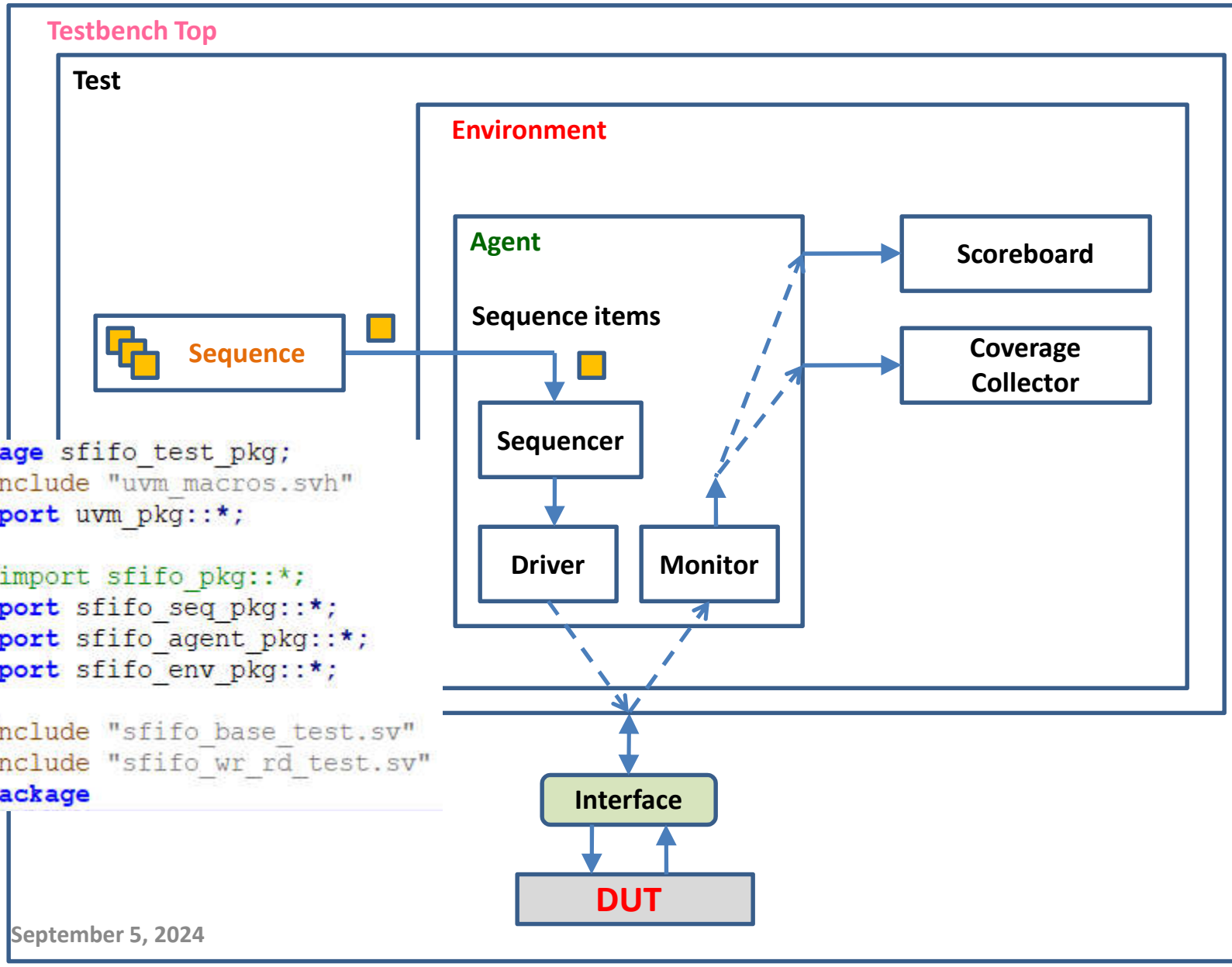

```
package sfifo_environment_pkg;

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

  // importing packages : agent, ...
  import sfifo_agent_pkg::*;

  // include Env components (include top Env files):
  `include "sfifo_scoreboard.svh"
  `include "sfifo_coverage.svh"

  // include Env file:
  `include "sfifo_environment.svh"

endpackage
```

Sequence Package




```
package sfifo_sequence_pkg;

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

  // importing packages : agent, ...
  import sfifo_agent_pkg::*;

  // include Sequence file:
  `include "sfifo_sequence.svh"

endpackage
```

Test Package




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

  //import sfifo_pkg::*;
  import sfifo_seq_pkg::*;
  import sfifo_agent_pkg::*;
  import sfifo_env_pkg::*;

  `include "sfifo_base_test.sv"
  `include "sfifo_wr_rd_test.sv"
endpackage
```

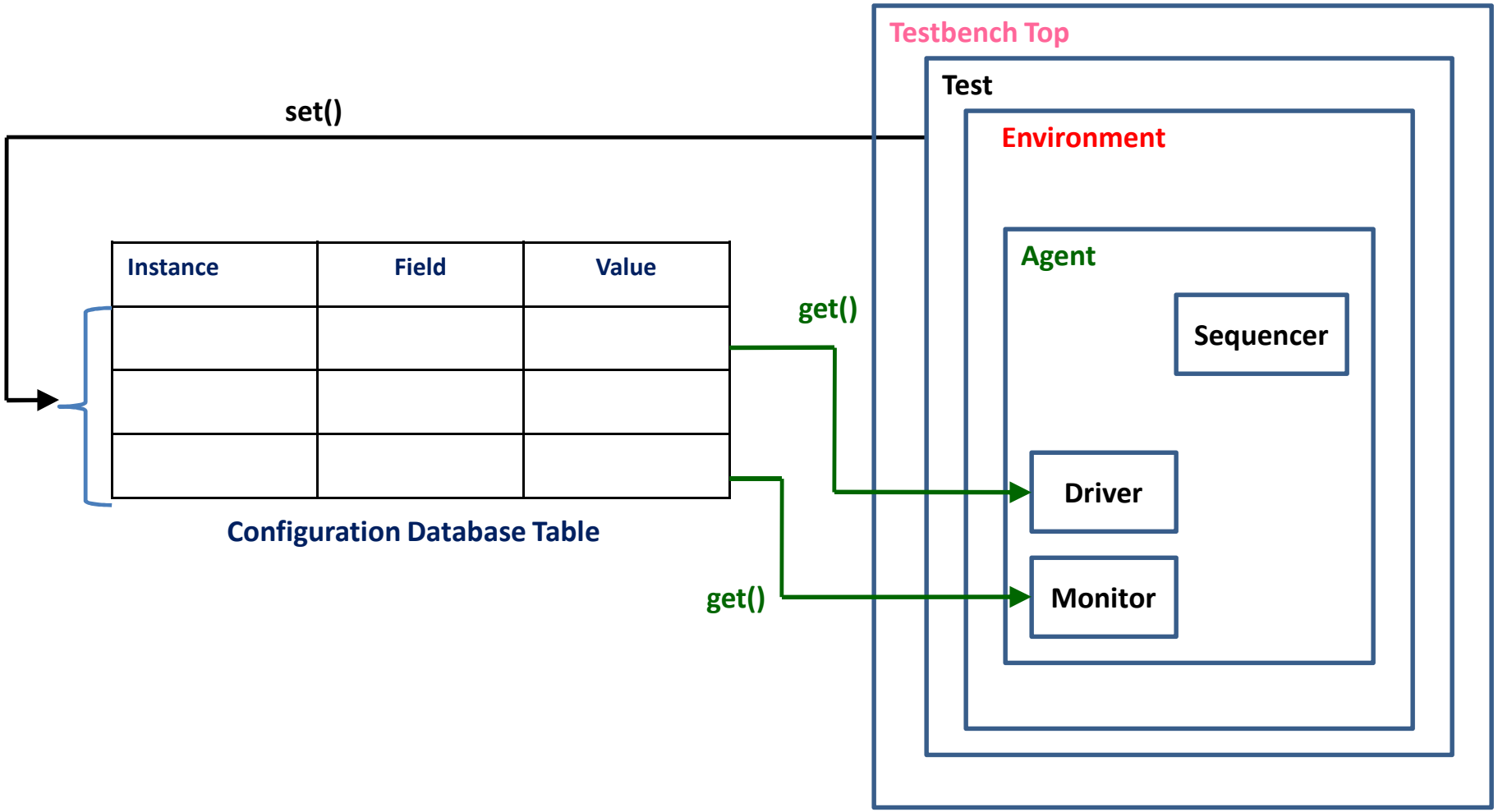
Configuration Implementation

Configuration Database

- **Configuration Database**
  - a *collection of information* that is stored and accessed on demand
  - basically acts as a *repository*
    - so that when the run time comes,
      - certain portions of the UVM testbench can be obtained from the **configuration database**
        - » and used to build the structure

- When items are placed in the **configuration database** with a **set()** method (**uvm_config_db::set()**),
  - components in **lower levels** will call the **get()** method
    - in order to obtain the necessary parts
      - to build the verification framework.

Configuration Database (2)



API and Application

//set method

- uvm_config_db #(type)::set(context, instance_name, "field", value);

- uvm_config_db #(virtual sfifo_interface)::set(null, "*", "vif", tif);

//get method

- uvm_config_db #(type)::get(context, instance_name, "field", value)

- uvm_config_db #(virtual sfifo_interface)::get(this, "*", "vif", tif)

Sharing/Propagating a Virtual Interface

- To '**set**' an **interface** from **top level** into the **configuration database**
  - while simultaneously giving **interface** an identifying name,
    - officially referred to as the '**field name**',
- In this work, setting the **interface** in the **configuration database**
  - using an identifier '**vif**'.
- In tb_top module:

```
sfifo_interface tif (clk, reset); // test interface

sync_fifo dut(.clk(tif.clk),      // dut instantiation
              .reset(tif.reset),
              .input_data(tif.input_data),
              .wr_en(tif.wr_en),
              .rd_en(tif.rd_en),
              .full(tif.full),
              .empty(tif.empty),
              .output_data(tif.output_data));

initial begin
  uvm_config_db#(virtual sfifo_interface)::set(.cntxt(null),
                                               .inst_name("uvm_test_top.*"),
                                               .field_name("vif"),
                                               .value(tif));
  run_test();
end
```

Sharing/Propagating a Virtual Interface (2)

- Since the tb_top module is not an *uvm_component*,
  - "null" is specified as the context argument

```
sfifo_interface tif (clk, reset); // test interface

sync_fifo dut(.clk(tif.clk),      // dut instantiation
              .reset(tif.reset),
              .input_data(tif.input_data),
              .wr_en(tif.wr_en),
              .rd_en(tif.rd_en),
              .full(tif.full),
              .empty(tif.empty),
              .output_data(tif.output_data));

initial begin
  uvm_config_db#(virtual sfifo_interface)::set(null, "*", "vif", tif);
  run_test();
end
```

Sharing/Propagating a Virtual Interface (3)

- Later, use the '**field name**' ('**vif**') to retrieve that **interface** in the **driver** and **monitor**
  - to connect to the **DUT**
    - by calling the **get()** method (**uvm_config_db::get()**)

- sfifo_driver.svh

```
virtual function void build_phase(uvm_phase phase);
  super.build_phase(phase);
  if(!uvm_config_db#(virtual sfifo_interface)::get(this, "", "vif", vif))
    `uvm_fatal("Driver: ", "No vif is found!")
endfunction
```

- sfifo_monitor.svh

```
virtual function void build_phase(uvm_phase phase);
  super.build_phase(phase);
  item_got = sfifo_seq_item::type_id::create("item_got");
  if(!uvm_config_db#(virtual sfifo_interface)::get(this, "", "vif", vif))
    `uvm_fatal("Monitor: ", "No vif is found!")
endfunction
```

Factory Implementation

Factory Implementation – API

- Register **components** and **objects** with the **factory**

```
`uvm_component_utils(component_type)
```

```
`uvm_object_utils(object_type)
```

- Construct **components** and **objects** using create not ***new***
  - **components** should be created during build phase of parent

```
component_type::type_id::create("name",this);
```

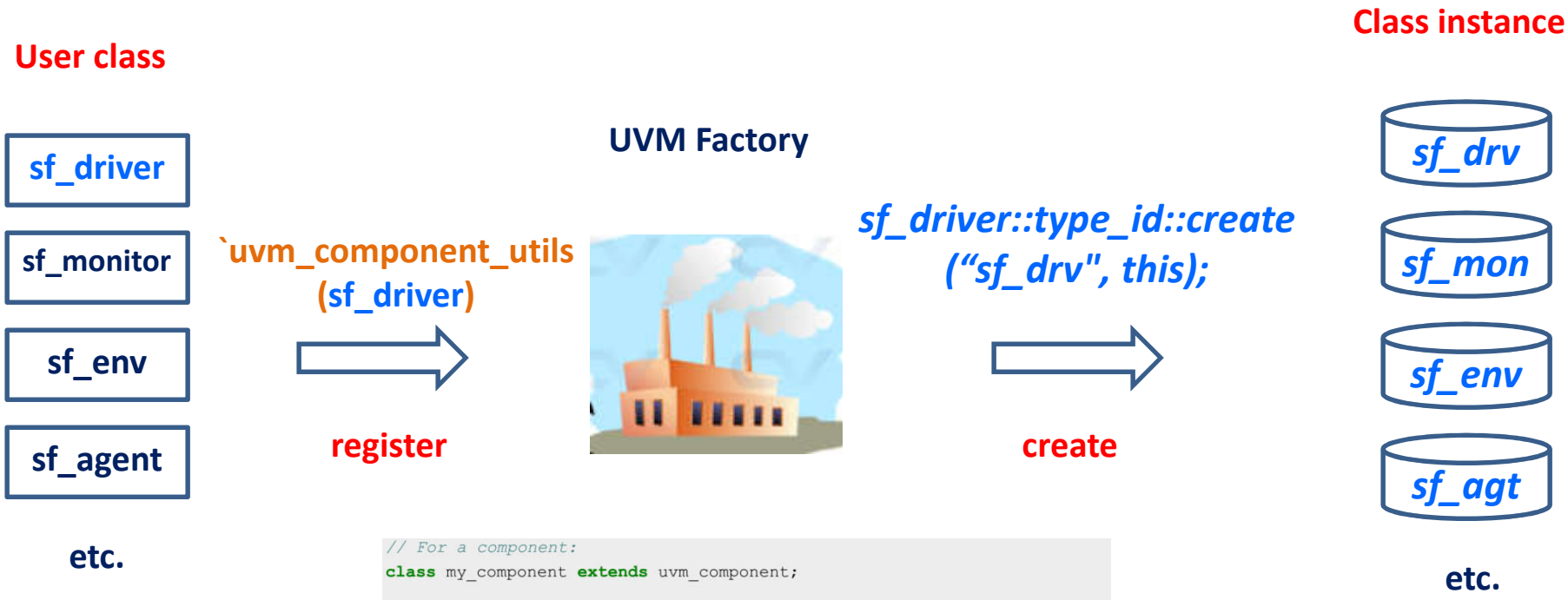
```
object_type::type_id::create("name",this);
```

Factory Implementation

- In order to register a **class** in **factory**, two macros are used:
  - **`uvm_component_utils**:
    - This macro registers *classes names* which are derived from **uvm_component** base class.
  - **`uvm_object_utils**:
    - This macro register *class names* that are derived from **uvm_transaction**, **uvm_sequence**, etc.
- When object of a class (component class) is created,
  - it should be created using the **factory** instead of *new()*.

- Creates the object (in **sfifo_test** class):
  - ***f_env = sfifo_environment::type_id::create("f_env", this);***
    - *f_env* is the handle to the component being constructed
    - *sfifo_environment* is the component's class name
    - *type_id* object is a singleton design pattern
    - *create* is a static method inside static *type_id* object.

Factory Implementation (2)




```
// For a component:
class my_component extends uvm_component;

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

// For an object:
class my_item extends uvm_sequence_item;

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

Factory Implementation (3)

sfifo_test class

```
sfifo_sequence f_seq;
sfifo_environment f_env;
virtual function void build_phase(uvm_phase phase);
  super.build_phase(phase);
  f_seq = sfifo_sequence::type_id::create("f_seq", this);
  f_env = sfifo_environment::type_id::create("f_env", this);
endfunction
```

sfifo_environment class

```
sfifo_agent f_agt;
sfifo_scoreboard f_scb;
sfifo_coverage f_cov;
virtual function void build_phase(uvm_phase phase);
  super.build_phase(phase);
  f_agt = sfifo_agent::type_id::create("f_agt", this);
  f_scb = sfifo_scoreboard::type_id::create("f_scb", this);
  f_cov  = sfifo_coverage::type_id::create("f_cov", this);
endfunction
```

sfifo_agent class

```
sfifo_sequencer f_seqr;
sfifo_driver f_dri;
sfifo_monitor f_mon;
virtual function void build_phase(uvm_phase phase);
  super.build_phase(phase);
  if(get_is_active() == UVM_ACTIVE) begin
    f_seqr = sfifo_sequencer::type_id::create("f_seqr", this);
    f_dri = sfifo_driver::type_id::create("f_dri", this);
  end
  f_mon = sfifo_monitor::type_id::create("f_mon", this);
endfunction
```

Factory Constructor Template

- In order to support deferred construction during the **build phase**,
  - the **factory constructor** should contain **defaults** for the constructor arguments.
- This allows a *factory registered class* to be built inside the factory using the **defaults**.

```
// For a component:
class my_component extends uvm_component;

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

// For an object:
class my_item extends uvm_sequence_item;

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

Thank You