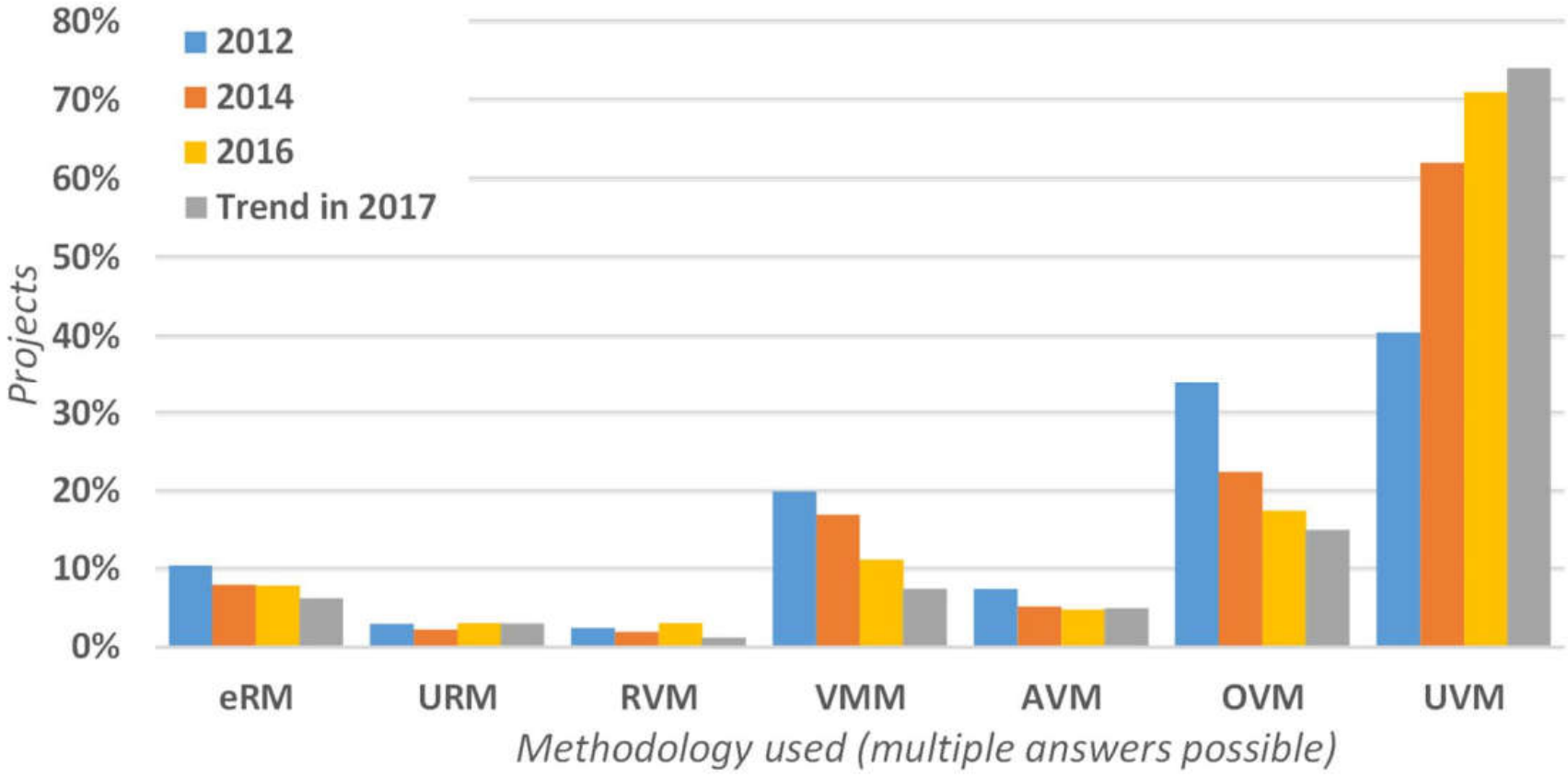
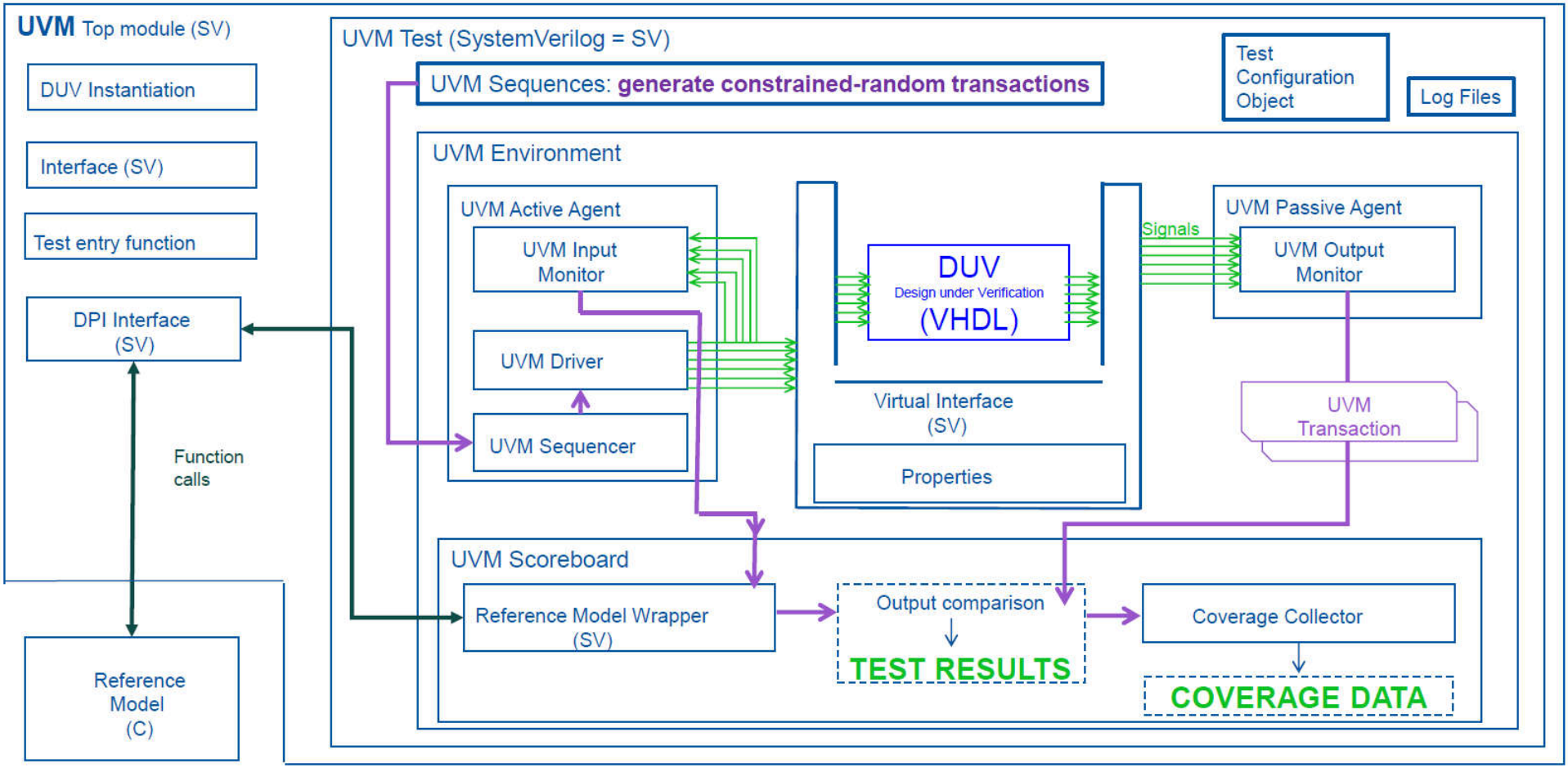
UVM (Universal Verification Methodology)

For SW Engineers

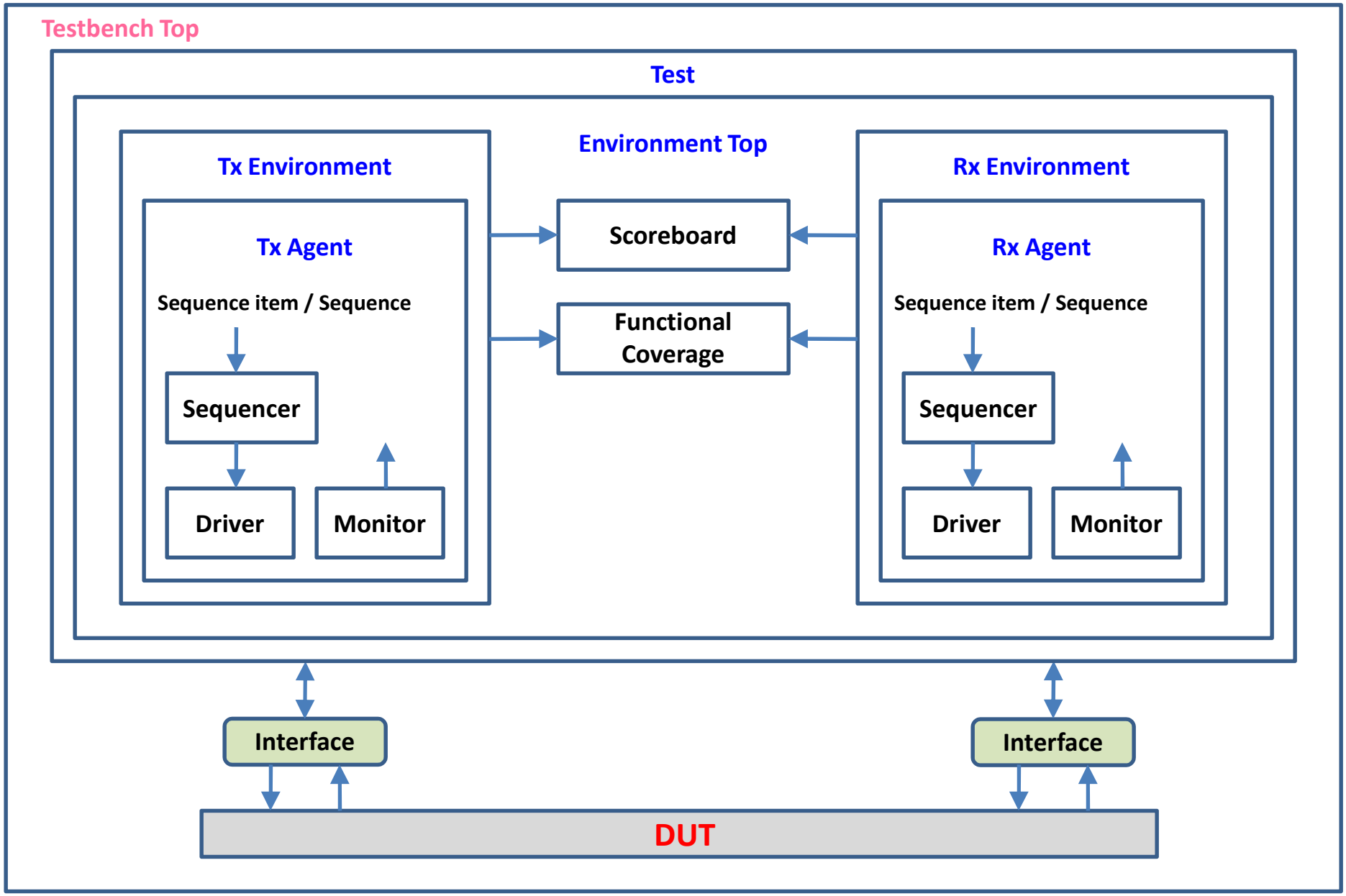
Tuan Nguyen-viet

ASIC/IC testbench methodology adoption trends

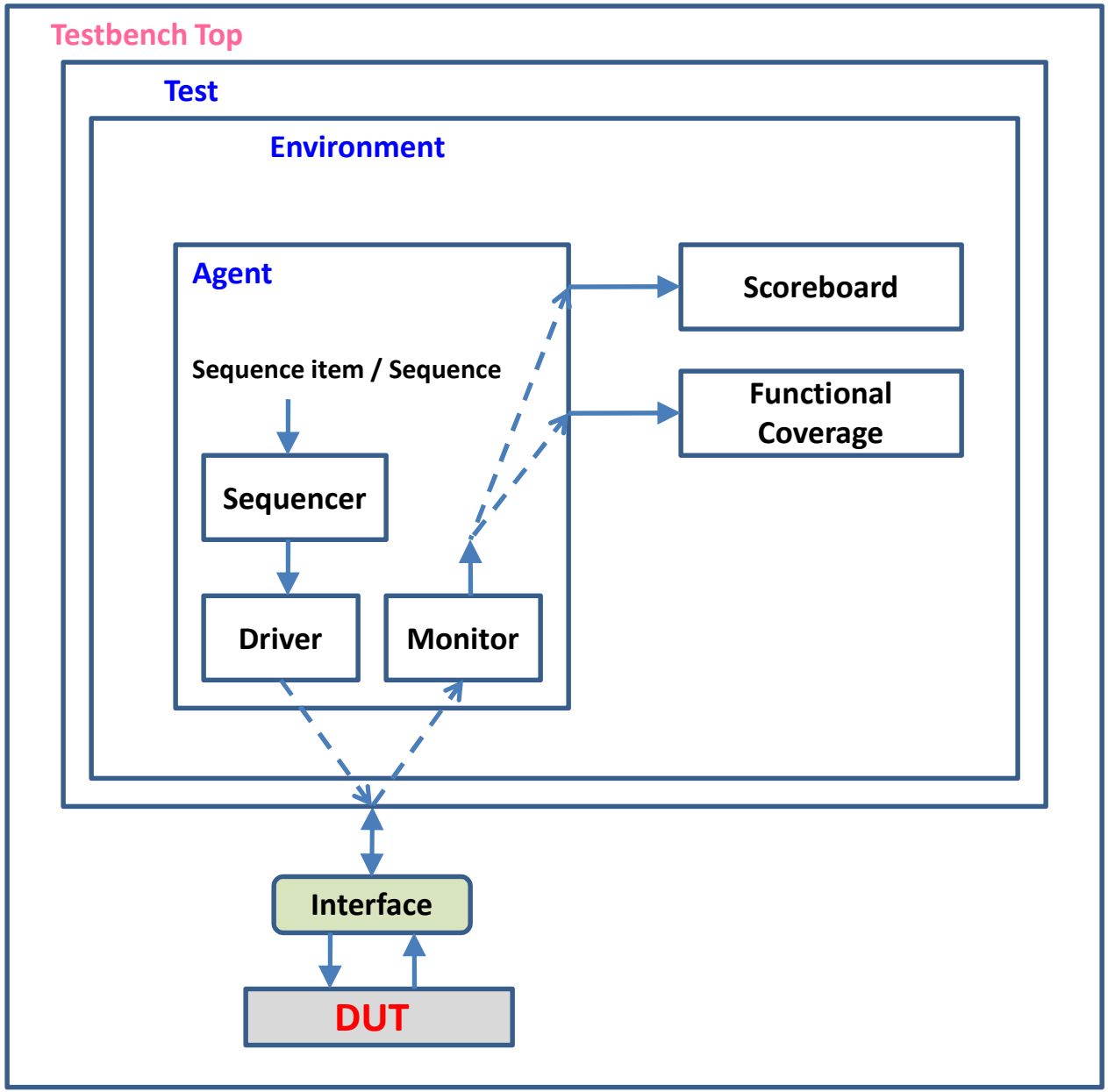



UVM Top module (SV)
DUV Instantiation
Interface (SV)
Test entry function
DPI Interface (SV)
Function calls
Reference Model (C)
UVM Test (SystemVerilog = SV)
UVM Sequences: generate constrained-random transactions
Test Configuration Object
Log Files
UVM Environment
UVM Active Agent
UVM Input Monitor
UVM Driver
UVM Sequencer
DUV
Design under Verification
(VHDL)
Virtual Interface (SV)
Properties
Signals
UVM Passive Agent
UVM Output Monitor
UVM Transaction
UVM Scoreboard
Reference Model Wrapper (SV)
Output comparison
TEST RESULTS
Coverage Collector
COVERAGE DATA

Key Components of a UVM Testbench



UVM - Simple Architecture w/ Single Agent



Hierarchy

- UVM testbench is built from dynamic objects,
  - that do not exist in memory before they are created,
    - a **static component** is needed for launching the simulation.

- The **static component** in UVM is a top level SystemVerilog module
  - that includes pin connections to the DUT
  - and starts the **test**,
    - which then configures the **environment**
      - and runs a **sequence of transactions** to the **DUT**.

Packages

- UVM testbench file hierarchy uses SystemVerilog ***packages***.
  - Packages are constructs that combine related declarations and definitions together in a common namespace that is a single compilation unit for the **simulator**.
- To access the namespace and the underlying definitions
  - the package must be *imported*.
- The usage of packages allows the testbench developer to organize the code and ensure consistent references to types and classes.

Packages (2)

- A package file should contain all the related class declaration files.

- > For a simple UVM testbench
  - a single package could contain all the definitions,
- < but in a large system level testbench
  - the declarations could be divided between multiple packages
  - so that there is
    - a separate package for every bus interface
    - and a number of packages for different types of test sequences
      - that contain all the declarations for running different tests.

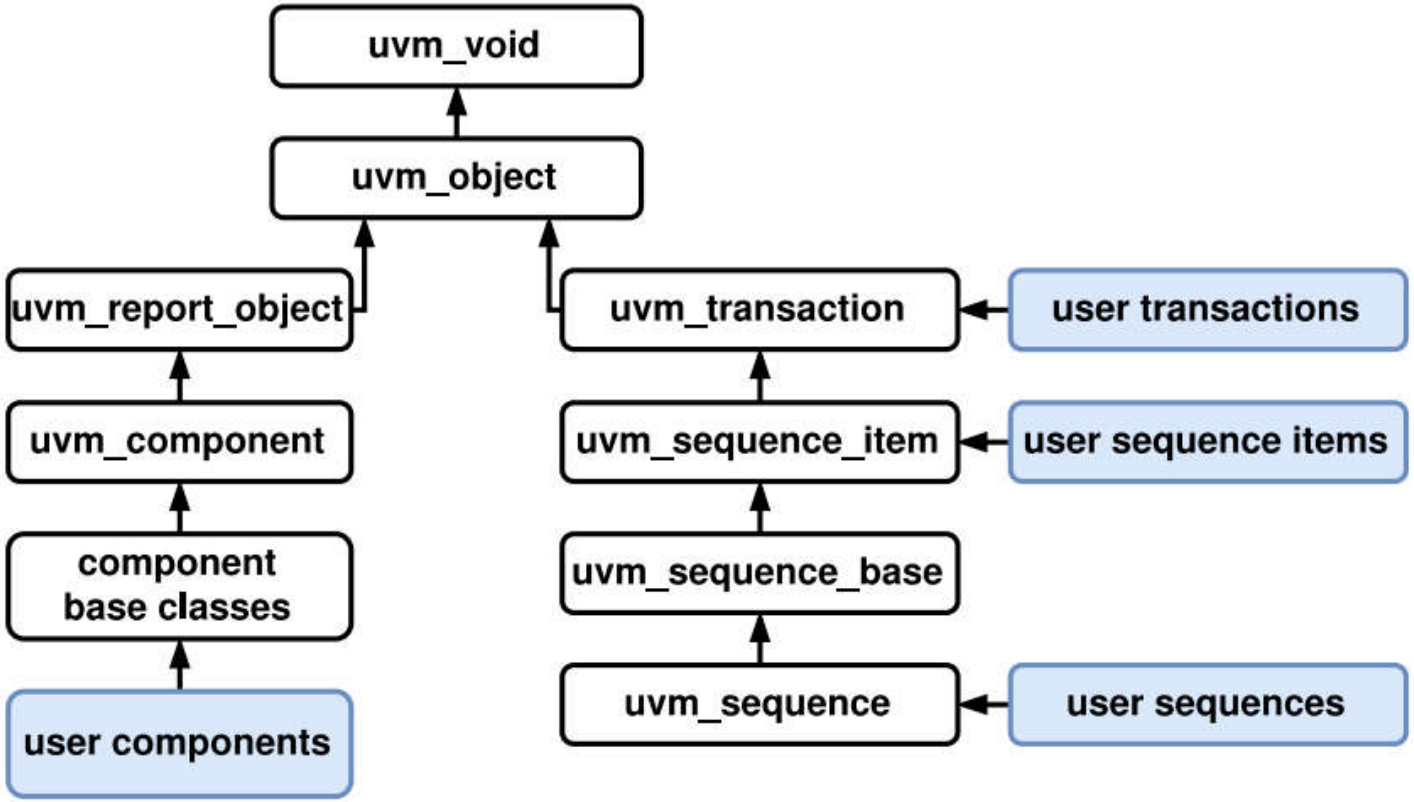
Packages (3)

- Instead of declaring all the classes directly in the package file,
  - the coding guidelines by Mentor Graphics state that
    - every class declaration should be in a separate file
    - and all the declaration files are included in the package
      - using a SystemVerilog include directive.
- The include directive instructs the **compiler** to insert the entire contents of a source file inside another file in place of the directive.
- The package should only contain the include directives for class declaration files.
- > The testbenches in the *first* exercises would be *simple*
  - so that the whole hierarchy can be declared in *one package*,
- < but in a more advanced additional exercise,
  - multiple sub-level packages could be introduced.

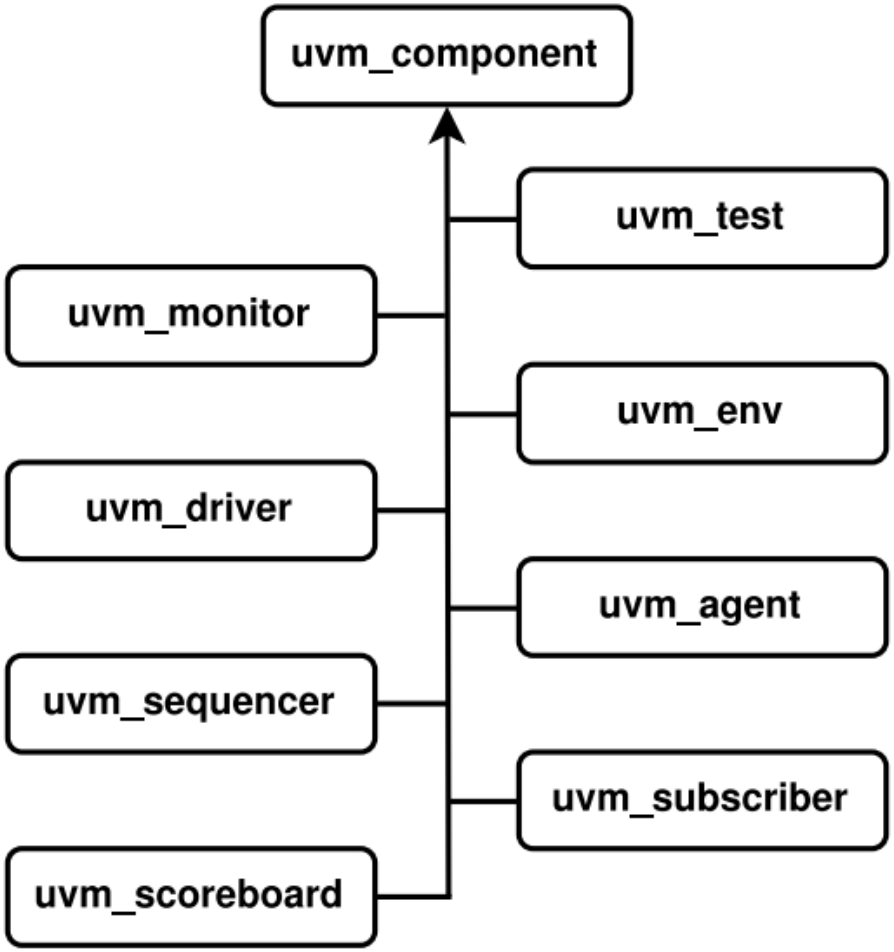
Objects and components

- *Object* is the basic building block in a UVM testbench
  - and all the objects are extended from the uvm_object base class.
- The primary role of the uvm_object base class
  - is to define the common methods for basic operations, e.g.
    - create and print,
      - that are used for every object.
- It also defines instance identification interfaces, e.g.
  - **name**
  - unique **id**.
- The most basic objects are *data packages* sent to the **DUT**
  - that are instantiated as *sequences of packages* to generate test input.

UVM base class hierarchy



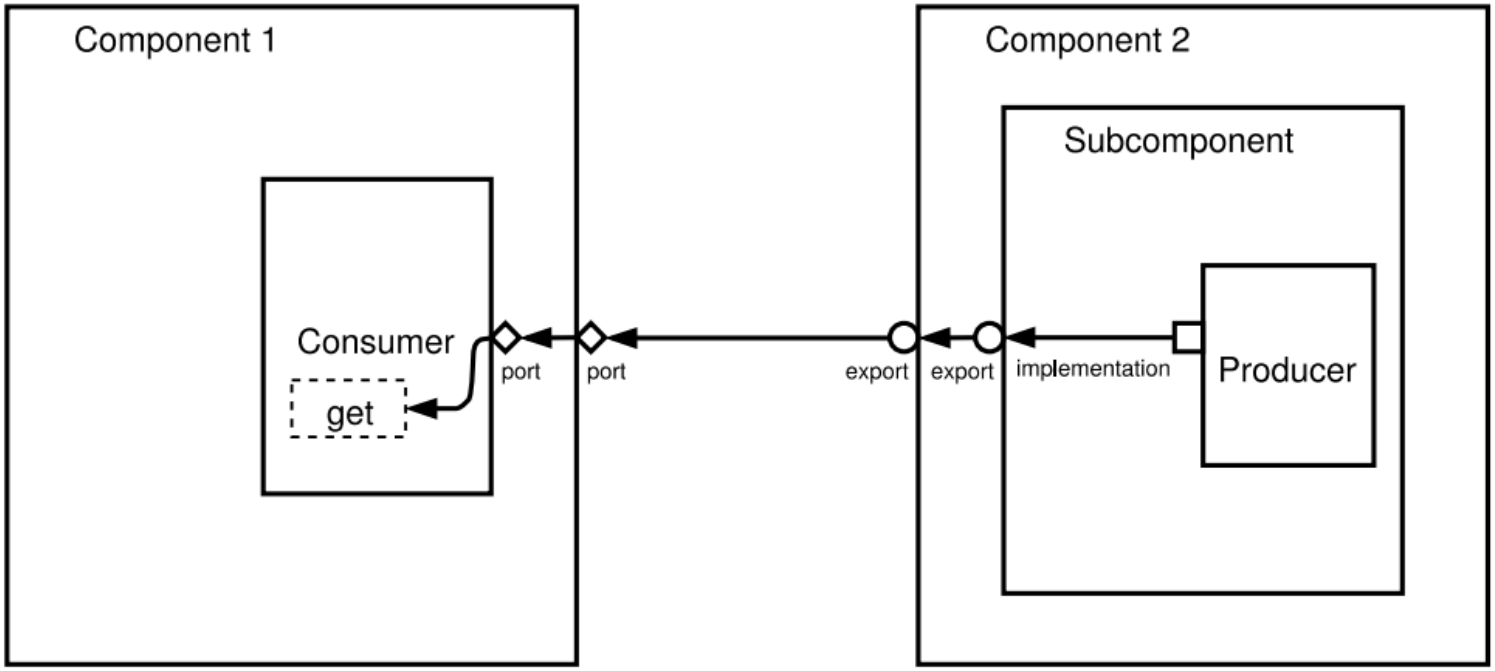
UVM component class hierarchy



UVM component class hierarchy (2)

- The components can communicate with each other
  - by delivering transaction level modeling (TLM) *transaction objects*
  - or by reading/writing the UVM configuration system.
- The TLM *transactions* are
  - delivered via channels between **ports** and **exports** in components
  - then connected to each other.
- A **port** initiate transaction requests.
- The **ports** are connected to implementations in components
  - that implement the *initiated* methods.
- **Exports** are channel items
  - that forward an implementation to be connected by the **port**.

UVM component class hierarchy (3)



Macros and methods

All the reporting in UVM should be done using ***reporting macros***

- A *sformatf* method is used to format the info message
  - using the syntax similar to printf function in the C language.
- There are also similar macros for errors and fatal errors.
  - **`uvm_info** (…)
  - **`uvm_warning** (…)

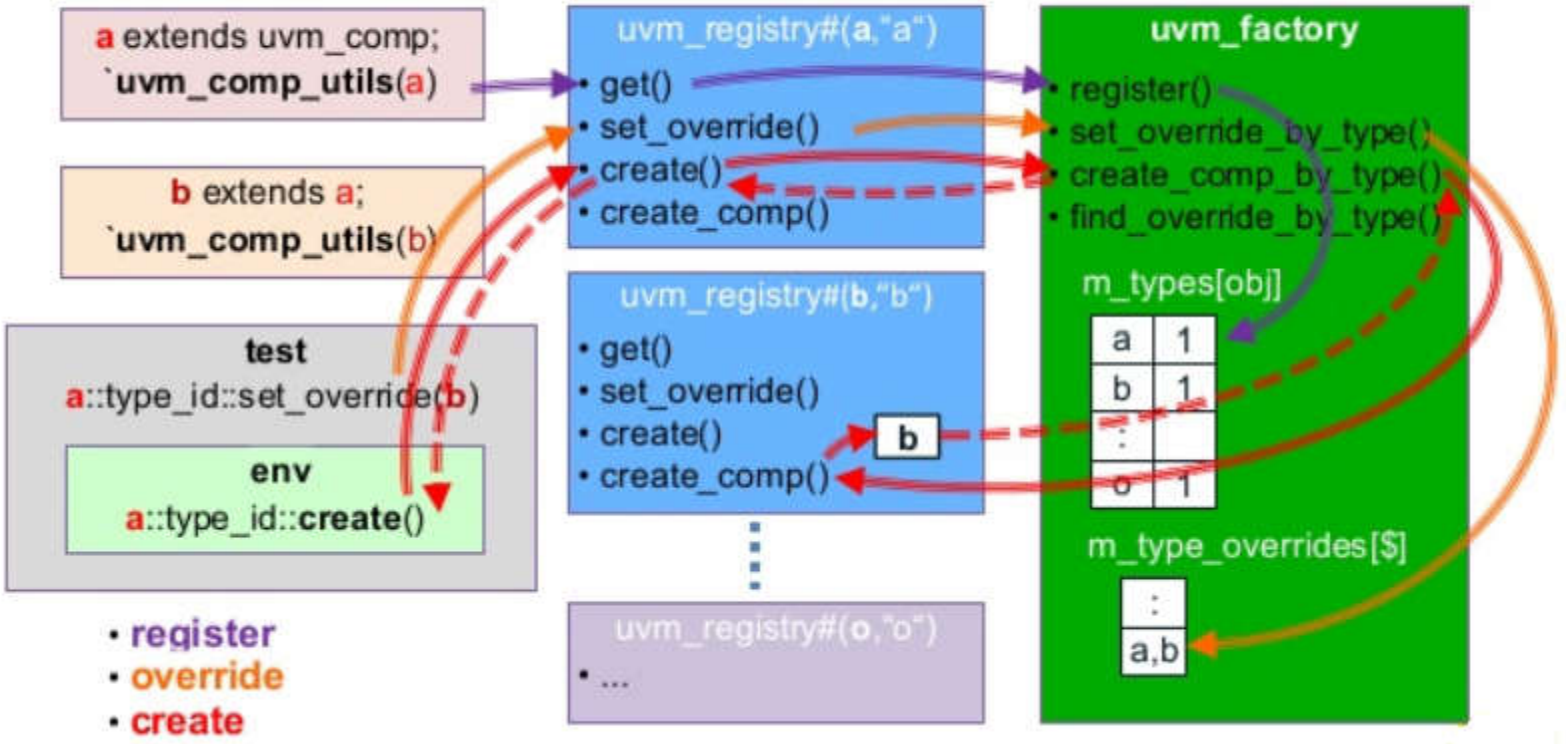
Other important macros are the ***factory registration macros***

- UVM factory
  - a database or a lookup table
  - stores **user-defined classes** (extended from UVM classes)
- The ***factory***
  - a class internal to the UVM mechanisms,
  - takes care of creating UVM objects and components
  - maintains a list of every instantiation done in the testbench.
- All the objects and components should be registered to *the factory*

Macros and methods (2)

- In order to register a class in *factory*, two macros are used:
  - **`uvm_component_utils(a class)**:
    - This macro registers class names
      - which are derived from uvm_component base class.
  - **`uvm_object_utils(a class)**:
    - This macro registers class names
      - that are derived from uvm_transaction, uvm_sequence, etc.

UVM factory



REF: https://www.asictronix.com/uvm-factory-and-uvm-configuration/

UVM factory (2)

- The purpose of the registration macro is to help *the factory* to keep a record of every object and component in the testbench.
  - The classes can be later substituted with *another compatible* class
    - by using *the factory* without changing the underlying component hierarchy code.

In addition to the registration macro,

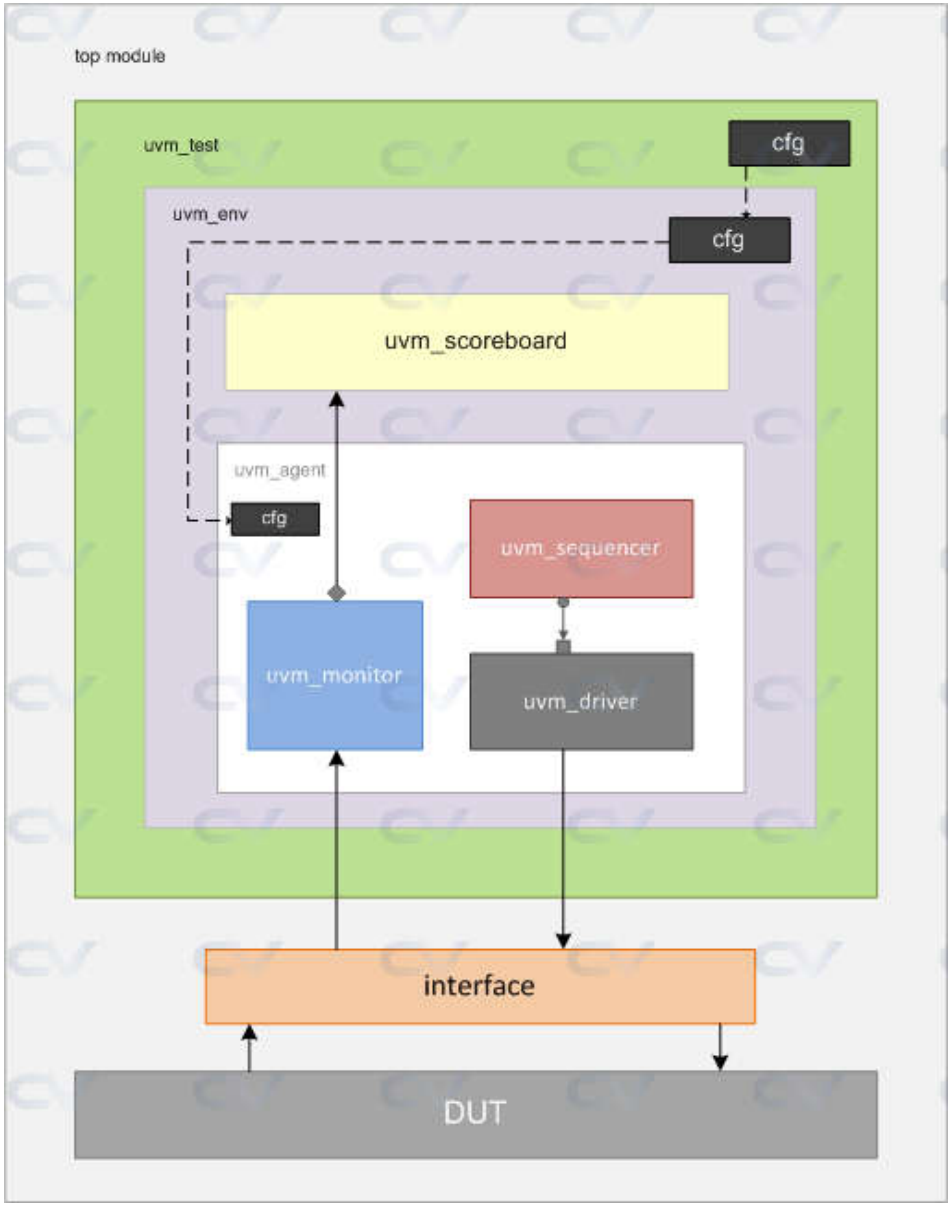
- the class instantiation should be done using a special factory method
  - instead of calling the constructor function directly.
- The factory method will call the constructor function of the classes,
  - but also performs additional procedures that are mandatory for the function of the UVM factory.
- The syntax for the factory method for instantiating an imaginary class **comp** :
  - **comp_h = comp::type_id::create("comp_h", this);**

UVM Configuration System

Configuration database

- Another important internal mechanism of UVM is the **configuration database**.
- The configuration database stores *variables* to be read in the components to allow communication across the testbench during runtime.
- In addition to the variable name and value,
  - a scope is set that dictates the *hierarchical path* to the component using the value.
- The configuration database can be written and read by every component
  - by using functions **set** and **get**.
- Usage of the configuration database enhances efficient reuse
  - by making the components in the testbench more *configurable*.

Configuration database (2)



REF: https://www.chipverify.com/uvm/configure-components

Configuration database (2)

- UVM manages the configuration database using a syntax called ***uvm_config_db***.

- *The syntax is as follows.*
  - ***set*** : " *Set* the database named *field_name* in *inst_name* located in *cntxt* to the value named value *!* "
  - ***get*** : " *Get* the database named *field_name* in *inst_name* located in *cntxt* as *value* !“

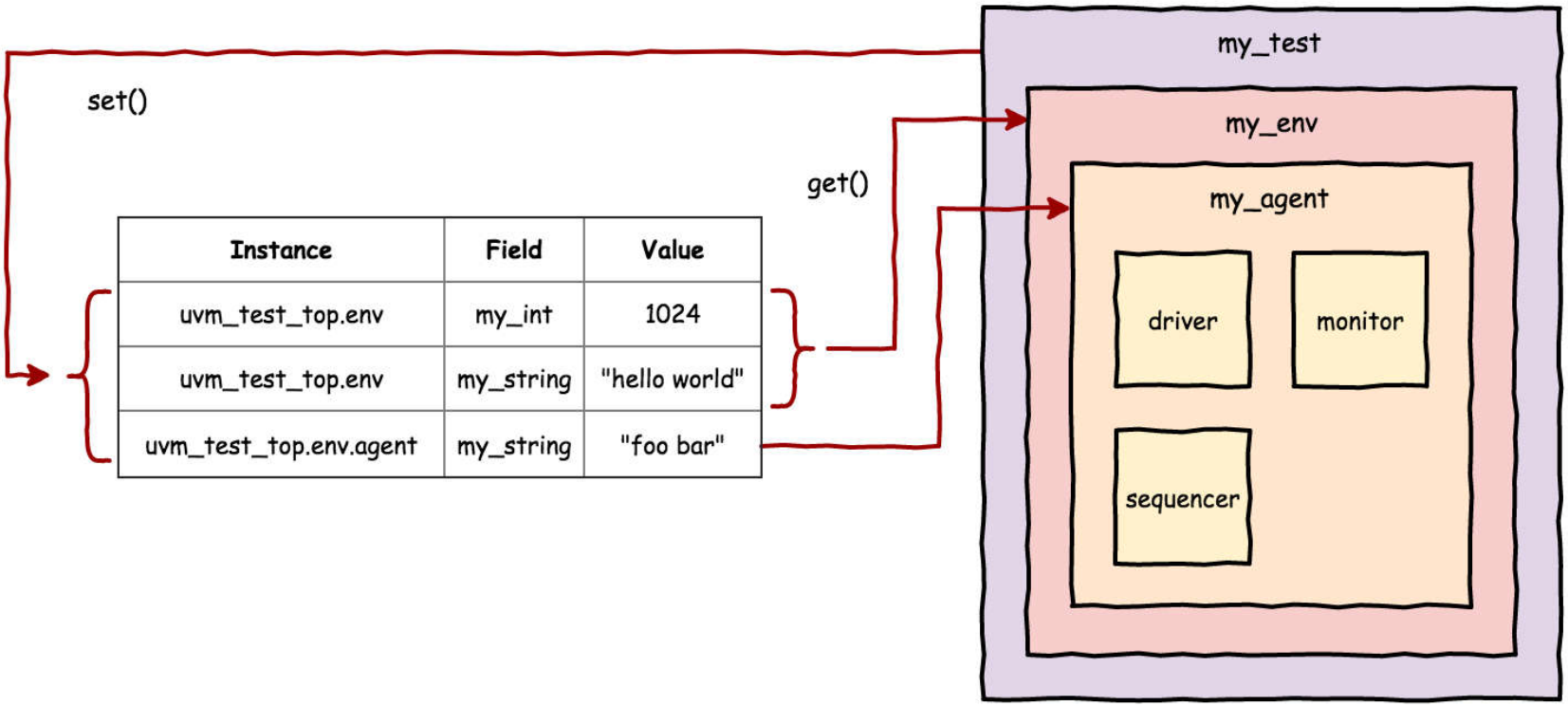
```
static function void set(uvm_componeny cntxt, string inst_name, string
    field_name, T value)
static function bit get(uvm_component cntxt, string inst_name, string
    field_name, inout T value)
```

- The first argument is the context (cntxt) which is **the starting point of the lookup search**.

Configuration database (3)

Argument	Description
Component Context (cntxt)	Specifies the scope of the configuration setting.
Instance Name (inst_name)	Identifies the specific instance of the component.
Field Name (field_name)	Specifies the configuration parameter we want to modify or add.
Value	Represents the new value to assign to the configuration setting.

Configuration database (4)



Configuration database (5)

```
// set configuration database of type int on env
  uvm_config_db#(int)::set(this, "env", "my_int", 1024);

  // set configuration database of type string on env
  uvm_config_db#(string)::set(this, "env", "my_string", "hello world");

  // set configuration database of type string on agent
  uvm_config_db#(string)::set(this, "env.agent", "my_string", "foo bar");
```

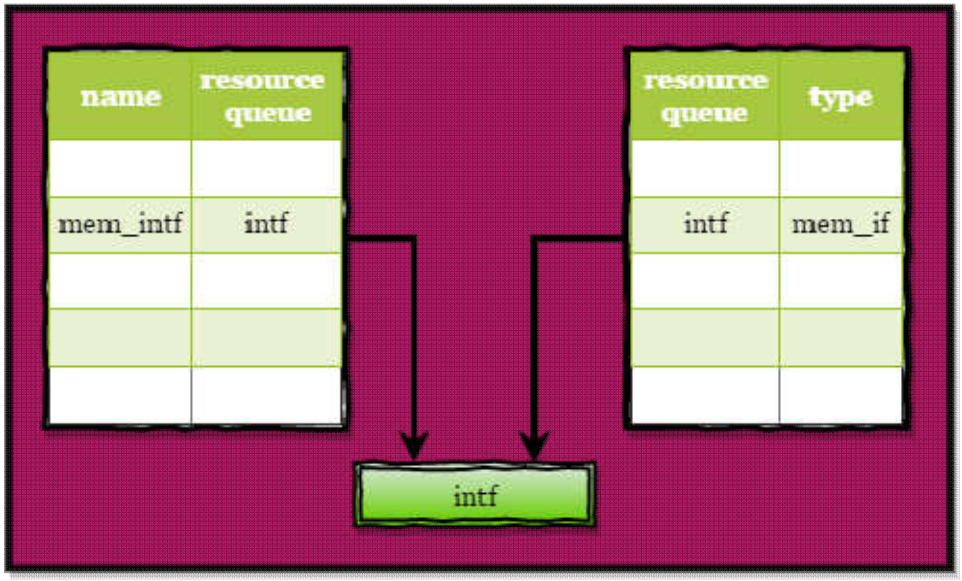
Configuration database (6)

void uvm_config_db#(type T = int)::set(uvm_component cntxt, string inst_name, string field_name, T value);

- Setting the interface handle intf, type mem_if, label mem_intf with global scope.

```
mem_if intf(clk,reset);  //interface instance
uvm_config_db#(virtual mem_if)::set(null,"*","mem_intf",intf);  //set method
```

- How a resource whose name is mem_intf and type is mem_if is stored in the pool.



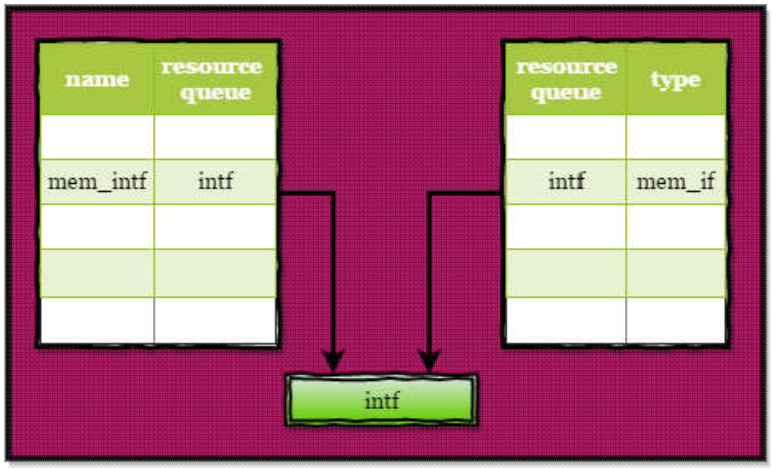
Configuration database (7)

bit uvm_config_db#(type T=int)::get(uvm_component cntxt, string inst_name, string field_name, ref T value);

- Using the get method to get a *virtual interface handle* from a database and assigns it to *mem_vif*.
  - If the get method fails, the fatal message will be displayed.

```
virtual interface mem_if mem_vif;    //virtual interface declaration
if( !uvm_config_db#(virtual mem_if)::get(this,"*", "mem_intf", mem_vif))
    `uvm_fatal(get_full_name(),{"virtual interface must be set for:",".mem_vif"} ); //get method
```

- How a resource whose name is mem_intf and type is mem_if is stored in the pool.



Thank You