**Session 2 :**

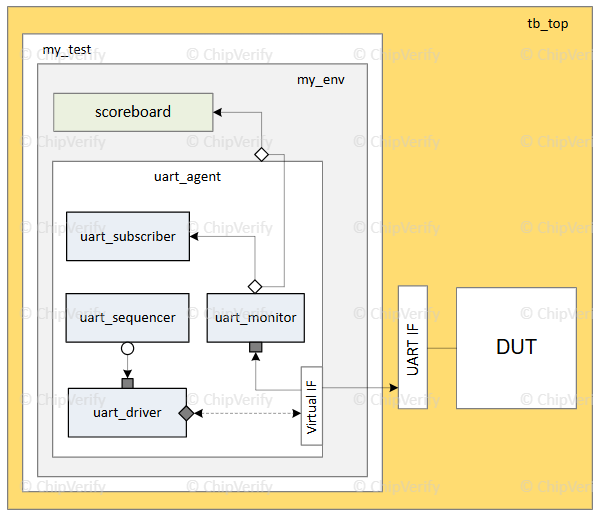
**What is Coverage Driven Verification (CDA) ?**

Coverage-Driven Verification (CDV) in UVM is a smart way to check if a chip design works correctly. Instead of writing lots of manual tests, UVM uses automatic test generators to create many different test cases. While the tests run, special “coverage monitors” keep track of which parts of the design have been tested and which have not. This helps engineers quickly see what’s left to test and make sure nothing is missed

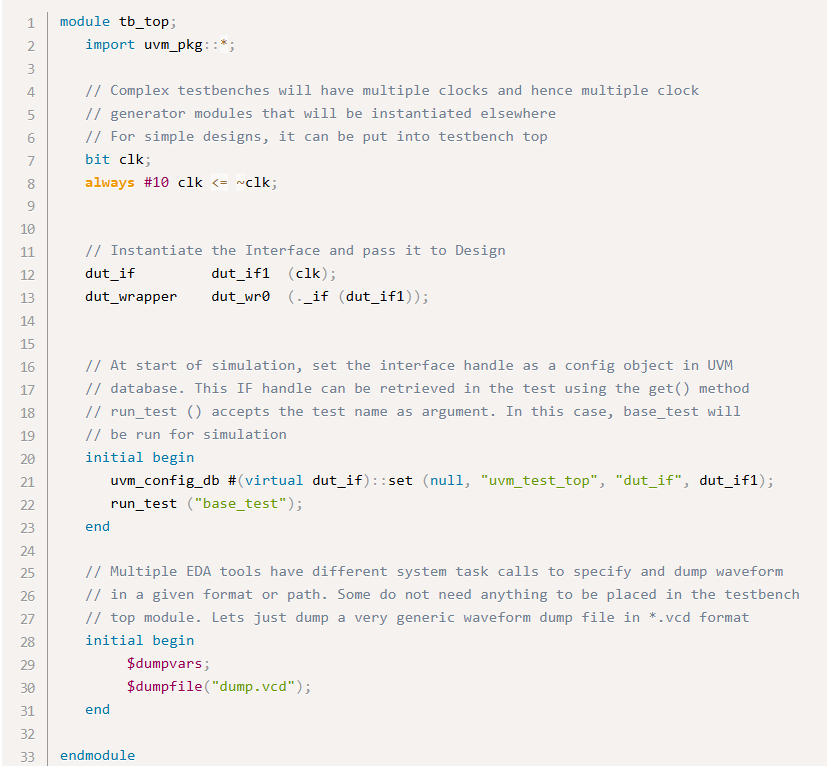
**UVM testbenches and enviornments**

**What is testbench top module ?**

All verification components, interfaces and DUT are instantiated in a *top level* module called testbench. It is a static container to hold everything required to be simulated and becomes the *root* node in the hierarchy. This is usually named tb or tb\_top although it can assume any other name.

****

Testbench for tb\_top :



1. Note the following :
2. tb\_top is a module and is a static container to hold everything else
3. It is required to import uvm\_pkg in order to use UVM constructs in this module
4. Clock is generated in the testbench and passed to the interface handle dut\_if1
5. The interface is set as an object in uvm\_config\_db via set and will be retrieved in the test class using get methods
6. The test is invoked by run\_test method which accepts name of the test class base\_test as an argument
7. Call waveform dump tasks if required

**Verification Model :**

**What is a Monitor ?**

A UVM monitor is responsible for capturing signal activity from the design interface and translate it into transaction level data objects that can be sent to other components.

In order to do so, it requires the following:

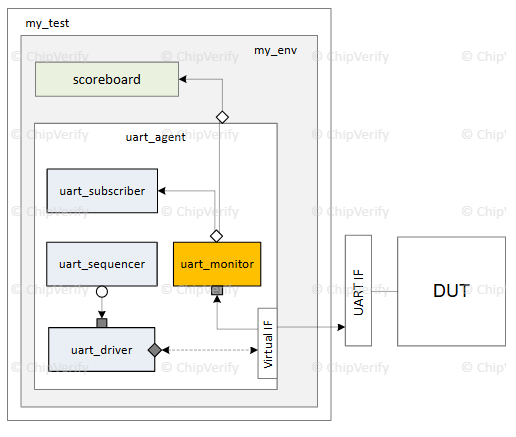
1. A virtual interface handle to the actual interface that this monitor is trying to monitor
2. [TLM Analysis Port](https://www.chipverify.com/uvm/uvm-tlm-analysis-port) declarations to broadcast captured data to others.

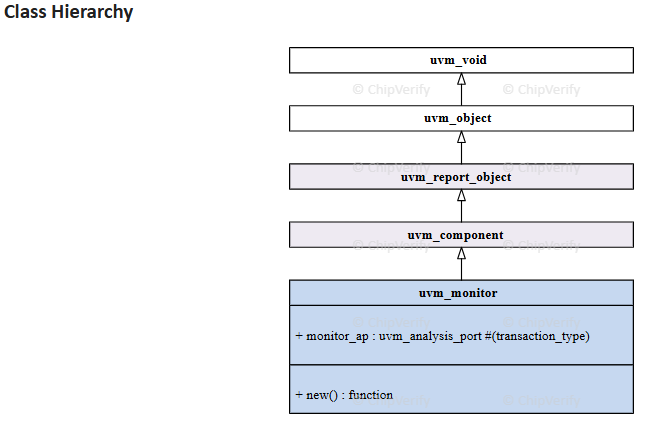
**What does a UVM monitor do ?**

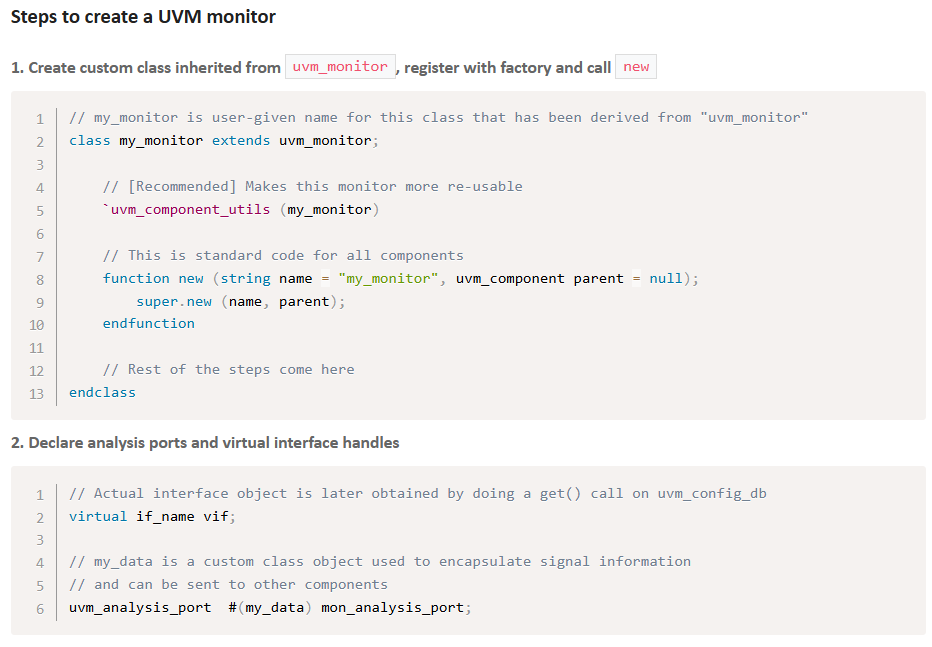
A UVM monitor is derived from uvm\_monitor base class and should have the following functions :

1. Collect bus or signal information through a virtual interface
2. Collected data can be used for protocol checking and coverage
3. Collected data is exported via an [analysis port](https://www.chipverify.com/uvm/uvm-tlm-analysis-port)

The UVM monitor functionality should be limited to basic monitoring that is always required.It may have knobs to enable/disable basic protocol checking and coverage collection. High level functional checking should be done outside the monitor, in a scoreboard.





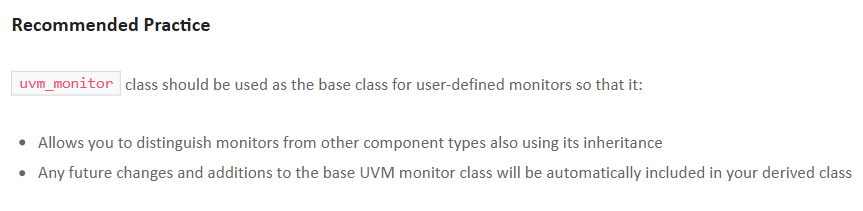




**UVM monitor example :**

****

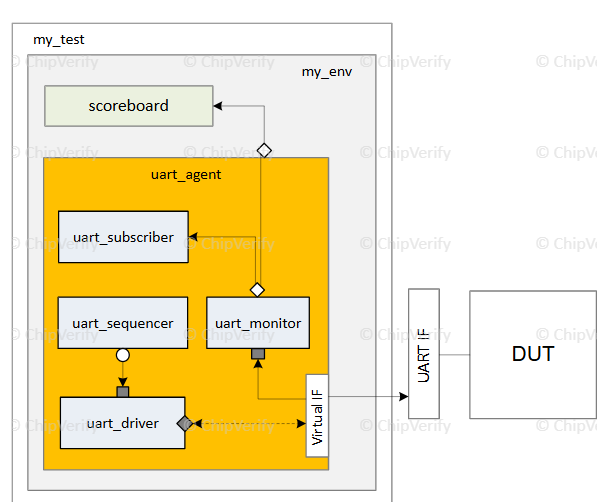
1. Note the following from the example above :
2. Monitor is extended from uvm\_monitor
3. Virtual interface handle is declared as **vif** and assigned from UVM database via uvm\_config\_db::get()
4. Additional knobs are provided for enabling/disabling protocol checker (*enable\_check*) and coverage (*enable\_coverage*)
5. Coverage group defined as **cg\_trans** and will be sampled during *run* phase
6. During run\_phase(), data from interface is captured into local class object, protocol check is performed when enabled, and coverage group is sampled when enabled
7. Data object class is broadcast to other verification components via the analysis port
8. The knobs can be disabled from the test by using UVM database.

****

**Agent**

**What is a UVM agent ?**

An **agent** encapsulates a [Sequencer](https://www.chipverify.com/uvm/uvm-sequencer), [Driver](https://www.chipverify.com/uvm/uvm-driver) and [Monitor](https://www.chipverify.com/uvm/uvm-monitor) into a single entity by instantiating and connecting the components together via TLM interfaces. Since UVM is all about configurability, an agent can also have configuration options like the type of UVM agent (active/passive), knobs to turn on features such as functional coverage, and other similar parameters.

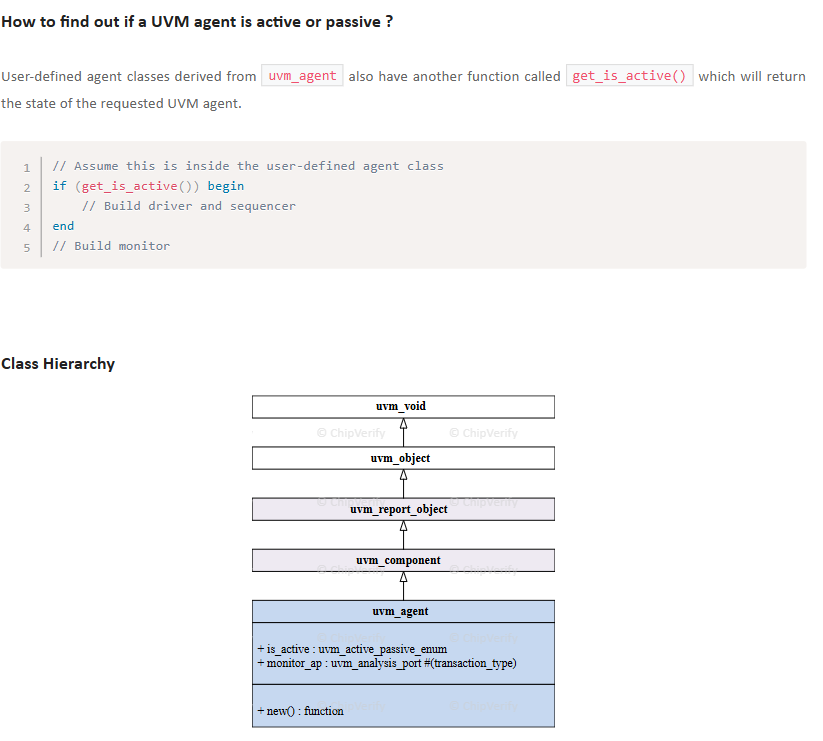


**Types of Agent**

**Active :** Instantiates all three components [Sequencer, Driver, Monitor] and Enables data to be driven to DUT via driver

**Passive :**

1. Only instantiate the monitor
2. Used for checking and coverage only
3. Useful when there's no data item to be driven to DUT

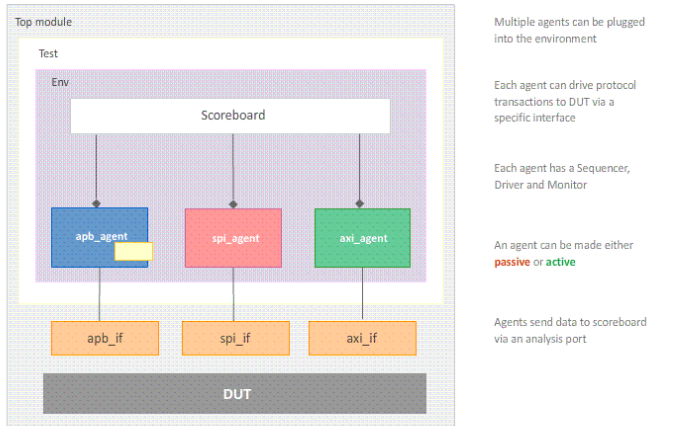


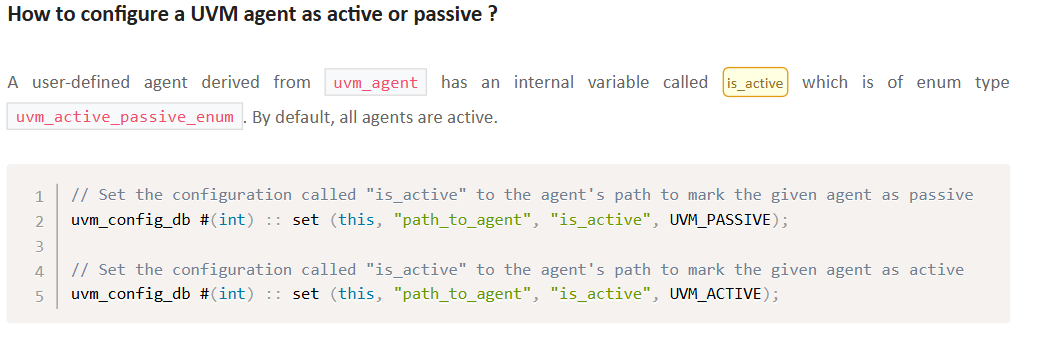




**What does a UVM agent do ?**

Usually, it makes sense to create an agent that provides protocol specific tasks to generate transactions, check the results and perform coverage. *For example*, a UVM agent can be created for the *WishBone* protocol whose sequencer will generate data items which can be sent to the driver. The driver then converts the data item class object into actual pin level signals and drive them to the DUT. The monitor may passively collect the outputs from the DUT, convert them back into another data item class object and distribute it among all the components in the testbench waiting for the item.

****

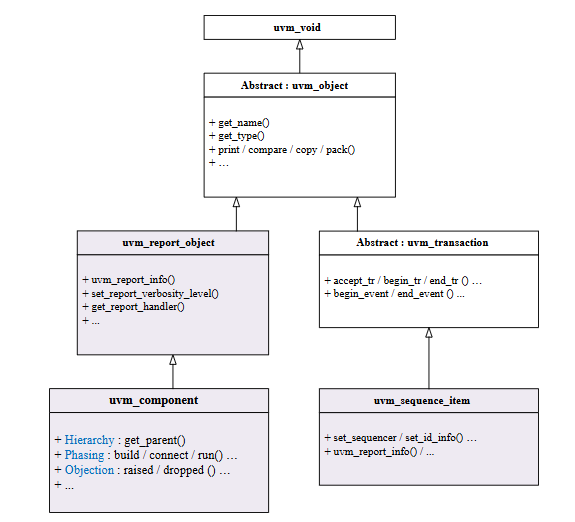
****

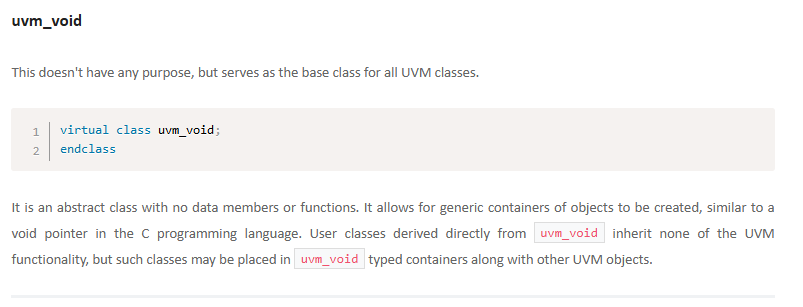
**UVM Class Library**

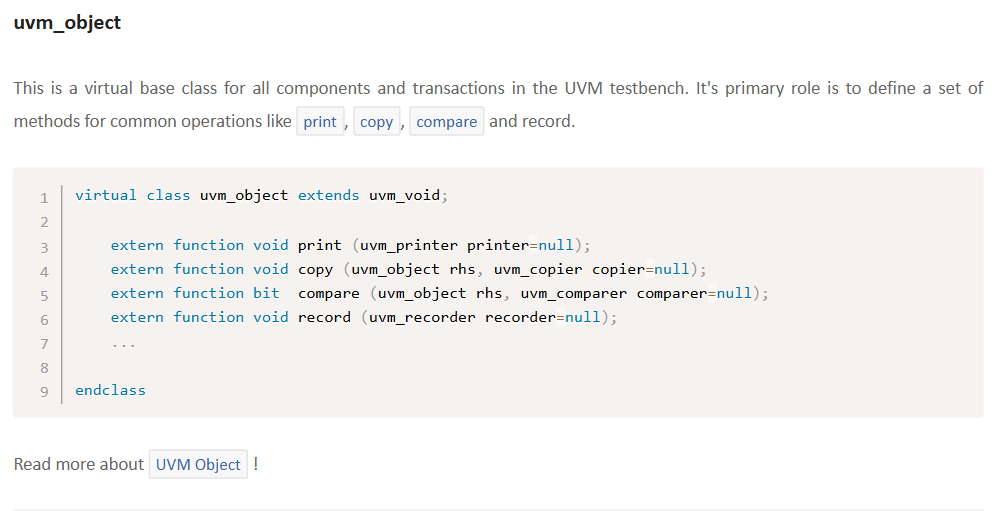
**Base Class**

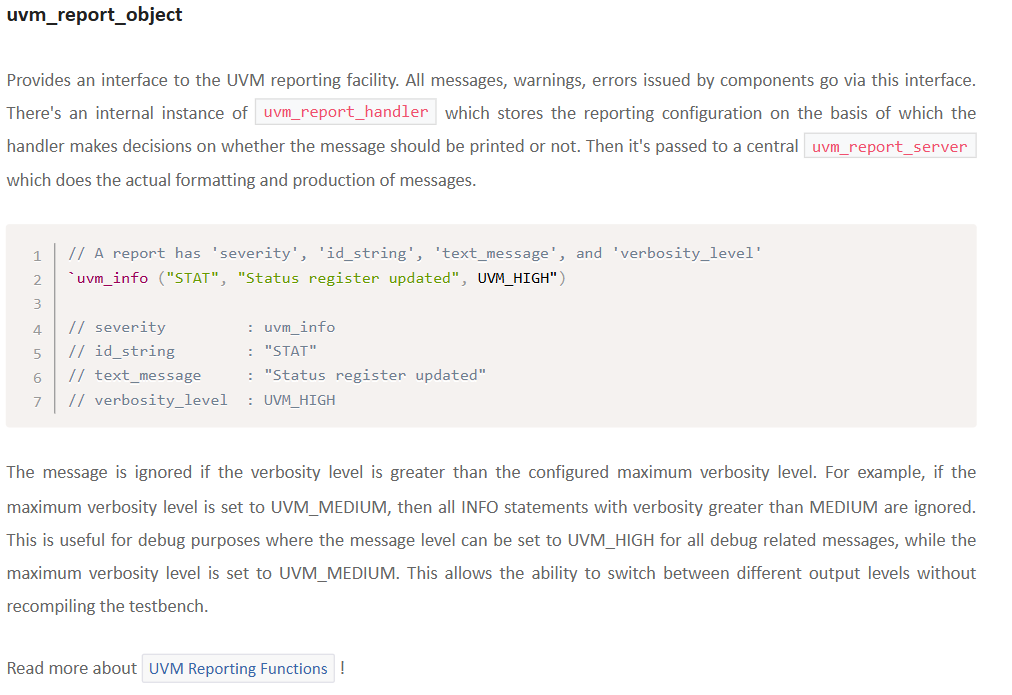
****

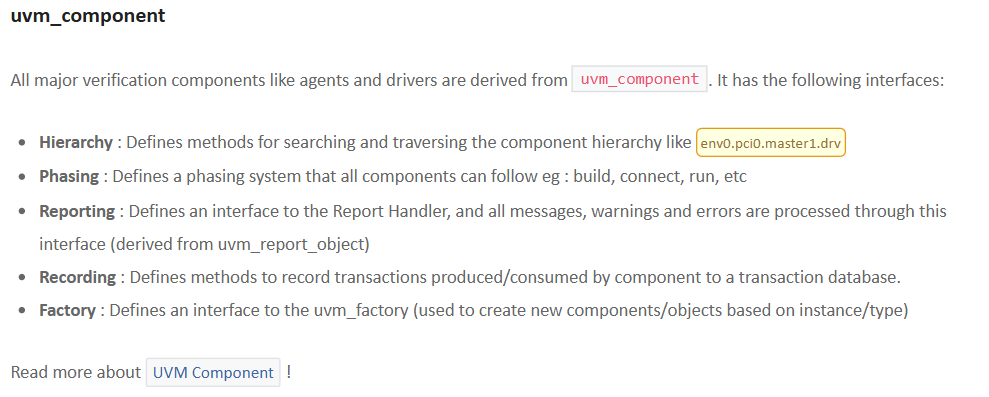
The basic building blocks for any verification environment are the components (drivers, sequencers, monitors ...) and the transactions (class objects that contain actual data) they use to communicate. From the UVM hierarchy, we can see that most of the classes in UVM are derived from a set of core classes that are described below.

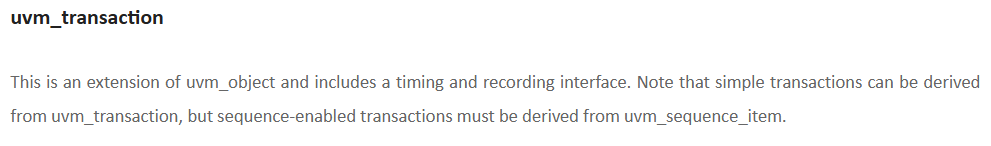
****

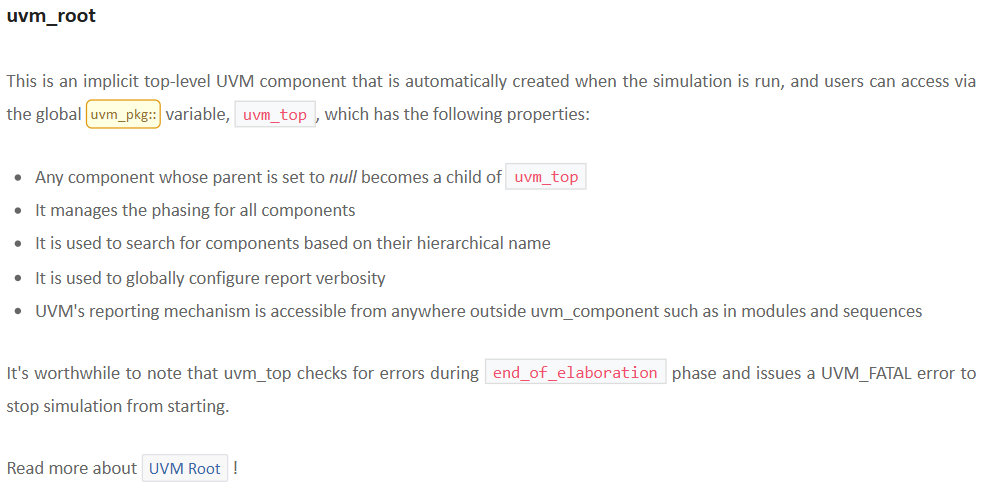
****

****

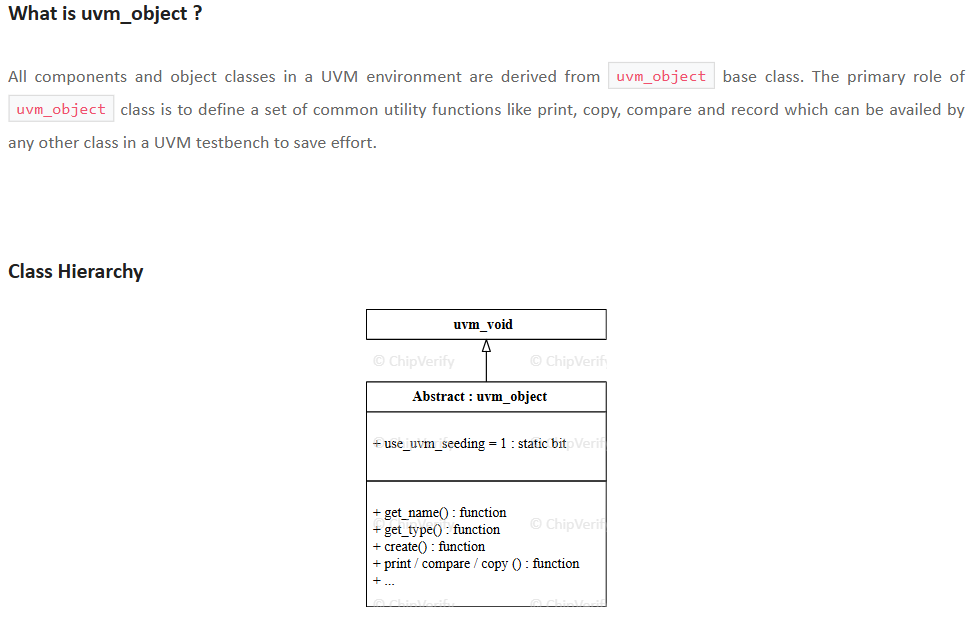
****

****

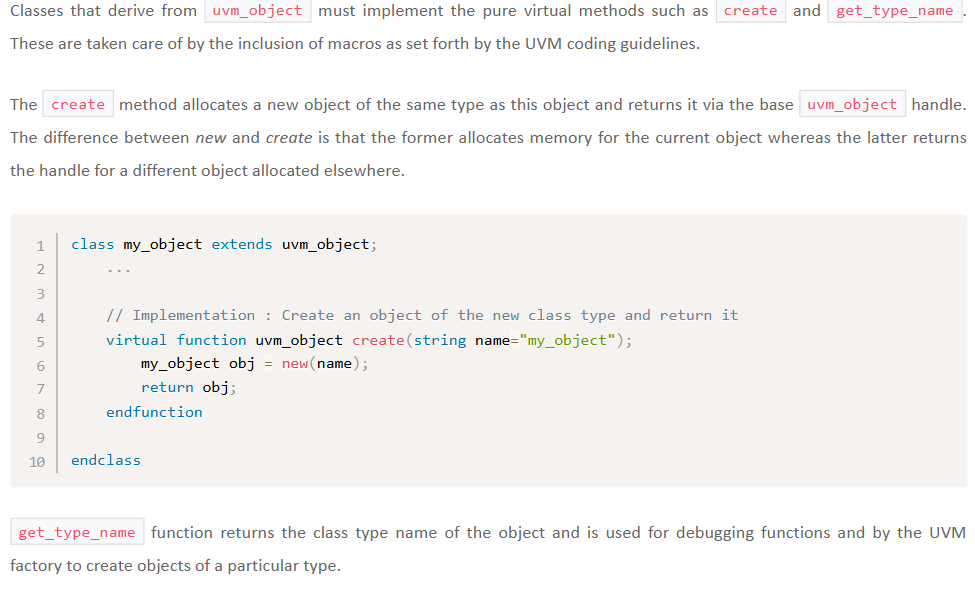
****

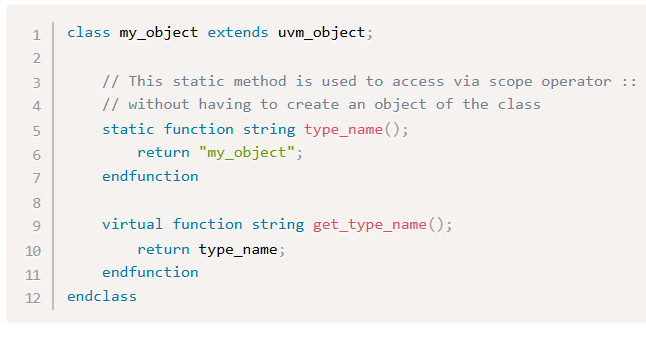
****

**Object :**

****

****

****

****