2019 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION INDIA

Using Software design patterns in testbench development for a multi-layer protocol

Pavan Yeluri, Sr. Asic Engnr., NVIDIA, Hyderabad, India (pyeluri@nvidia.com)

Ranjith Nair, Sr. Asic Manager, NVIDIA, Hyderabad, India (ranjithn@nvidia.com)

Abstract—

Modern day functional verification has become a very complex task with increasing design complexities and consumes almost 53% of the project time on an average as per recent studies [1]. Coupled with faster turnaround times, it becomes critical to create robust verification environments that are maintainable and reusable across different versions and configurations of IPs thus reducing the overall development life cycle. This paper describes how Software design patterns can be used for creation of a robust verification environment for a configurable multi-layer protocol. We have tried use these patterns to solve certain problems that we faced during the testbench development for a configurable MIPI DSI IP. Several design patterns are described along with the specific problem that they are used to solve.

Keywords— Design patterns; SV; UVM; functional verification; MIPI DSI;DPHY;CPHY

I. INTRODUCTION

Design patterns are a set of well-established techniques being used since long in the software world for code reusability, extensibility and maintenance. The hugely popular book “Design Patterns” [2] written by the authors who came to be known as ‘Gang of Four’(GoF) have given each of the patterns their definitive names and explained in detail their applications. System Verilog borrows heavily from software languages like C++ and Java and hence lends itself well for the application of design patterns. It is not a good design practice to solve every problem from first principles, instead, we should try to reuse the standard proven solution, if any, for a specific problem. Verification engineers do not do a good job of recording experience in testbench design for others to re-use. Design patterns help us solve this problem to a certain extent. UVM library code base already makes use of patterns like factory, observer etc., but creating any complex testbench involves extensive coding apart from making use of the UVM library. This makes the testbench development vulnerable to the expertise of the coder in making it robust and reusable. At any given point of time, a team may contain resources with various levels of coding expertise and may result in a testbench that is unstructured and hence neither flexible nor reusable. It bodes us well to incorporate the lessons learnt by decades of software design into functional verification and come up with a set of standard implementation procedures for commonly occurring scenarios thus making the testbench code more readable, reusable and maintainable.

Usage of a few design patterns was demonstrated in System Verilog [3] and an attempt has been made in [4] to use one of the design patterns called 'visitor', which adds new operation to an existing class without modifying it. Reference [5] explains a set of generic scenarios where design patterns are used. However, this paper attempts a comprehensive application of multiple design patterns in creation of a flexible testbench for a configurable multi-layer protocol IP.

II. VERIFICATION ENVIRONMENT DESCRIPTION

The Display Serial Interface (DSI) Specification [6] defines a high-speed serial interface between a peripheral, such as an active-matrix display module, and a host processor/display controller in a mobile device. By standardizing this interface, components may be developed that provide higher performance, lower power, less EMI and fewer pins than current devices, while maintaining compatibility across products from multiple vendors. DSI supports 2 different PHY layers called DPHY & CPHY the features of which keep changing across various versions.

DSI IP developed at NVIDIA is highly configurable and is developed to be able to support different PHY layers and various protocol specification versions.

A testbench was developed for single Protocol-PHY layer configuration (DSiv2.0-CPHYv1.2) and then it had to be made flexible to so that it can be extended to support DPHY. The testbench was then made adaptable enough to support any new Protocol-PHY layer configurations or any new versions of MIPI DSI spec with minimal changes to the existing code base. Following table lists a few differences between various configurations of DSI IP: -

Table-1 List of various configurations and modes

Protocol layer differences		
Feature	**CPHY**	**DPHY**
Data unit	16-bits	8-bits
Packet structures	Different for ESCAPE(ESC)/Low Power (LP) and HIGH SPEED (HS) mode operations	Same for both ESCAPE (ESC)/ Low Power (LP) and HIGH SPEED(HS) mode operations
Packet Header width	32-bits for ESC mode, 48-bits for HS mode	32-bits for both HS and ESC modes
Error detection and correction in Packet header	12-bit SSDC, 12-bit checksum and Packet Header replication for HS mode. 8-bit ECC for ESC mode	8-bit ECC for both HS and ESC modes
Number of lanes	2	5
LP mode substitution for blanking packets	Supports during HBP/HFP	Supports at the end of HS packets transmission
Non-word aligned Host packets	No limitation	Supports up to packets
EoTp	Not applicable	Applicable
EoT Sync error	Not applicable	Applicable
Version differences		
Feature	**DSI V1.1**	**DSIV1.2**
Direction	HS RX not supported	HS RX Supported
ALP mode	Not supported	Supported
Sync Symbol Sync Types	Not supported	Supported

Typically, to accommodate the changes specific to each of the configurations, all aspects of verification environment like sequences, transactions, constraints, scoreboard and functional coverage should be updated in the existing testbench and the resultant code becomes unstructured and difficult to maintain. The proposed approach tries to make this process more structured. These techniques can be applied to a basic testbench with minimal configurations as well in order to make it readable and extensible.

Following sections highlight a few instances where design patterns are used in the testbench and how these techniques are better compared to standard approach taken for the specific scenarios.

III. CODING CONSTRAINTS FOR VARIOUS CONFIGURATIONS AND MODES

Each of the PHY layers, versions and modes supported in the testbench needs to have a different set of constraints to be able to appropriately generate the stimulus. The example we take here is that of a simplified version of a base sequence, the variables of which need to be constrained differently specific to different PHY Layers, modes & versions for the corresponding stimulus generation. Below we try to explain the general approach taken for this scenario and how using a specific design pattern made the code more readable, structured and reusable.

Standard Approach

A derived sequence is created from the existing sequence and a new constraint is added to it or an existing constraint in base sequence is overridden. This derived sequence is then overridden throughout the test bench using *set_type_override* function. The main drawback with this approach is debugging becomes increasingly difficult as number of such overrides increases. Also, to cover the various combinations of the modes/PHY layers/versions a huge number of derived sequences are needed, and this leads to what is called ‘class explosion’ in software world.

A class explosion occurs when adding new functionality to existing class structure leads to huge class hierarchy. This is a very clumsy, unstructured approach to tackle this problem.

Proposed Approach: Use Decorator Pattern

"The Decorator design pattern attaches additional responsibilities to an object dynamically. Decorators provide a flexible alternative to sub classing for extending functionality" – Head first design patterns [7].

This is a structural pattern which provides us a mechanism to modify a class object by allowing us to add behavior to it without affecting the other objects of the class type. This technique makes the base class code future proof to a certain extent. The main advantage of this technique over inheritance is that the behavior can be added to only specific objects of the class dynamically. Using this pattern, we can add the constraints specific to each mode as a 'layer' over the existing constraints. In a way, we are 'decorat'ing the existing base class with additional functionality appropriately as per our needs. Given below are the components of this pattern as per [7]:

1) Base sequence class: - This is the base sequence that contains the variables on which various constraints are applied through the test case over the course of simulation. Figure 1 below contains to the code specific to this class.

```
// DSI Base sequnece
class dsi_base_seq extends uvm_sequence;

  //Class variables/properties to be constrained
  rand bit[3:0] num_of_lanes;
  rand int num_of_pkts;
  rand bit hs_rx_mode;
  typedef enum {END_OF_HS, HBP_OR_HFP} lp_mode_substitution;
  rand lp_mode_substitution_e;
  rand bit alp_mode;
  rand bit[27:0] sync_type;

  `uvm_object_utils_begin (dsi_base_seq)
    `uvm_field_object (num_of_lanes, UVM_ALL_ON|UVM_DEEP)
    `uvm_field_object (num_of_pkts, UVM_ALL_ON|UVM_DEEP)
    `uvm_field_object (lp_mode_substitution_e, UVM_ALL_ON|UVM_DEEP)
    `uvm_field_object (alp_mode, UVM_ALL_ON|UVM_DEEP)
    `uvm_field_object (sync_type, UVM_ALL_ON|UVM_DEEP)
  `uvm_object_utils_end

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

  //Ideally Constraints would be written here

  virtual task body();
  //Logic specific driving stimulus
  endtask: body
endclass: dsi_base_seq
```

Figure 1. Base sequence class

2) Base Decorator class: - This class is a wrapper class over the base sequence class and sets up the infrastructure to be used by the 'concrete decorator' classes (shown in Figure 3). Figure 2 below refers contains the code specific to this class.

```
//Note that Decorator both 'has-a' & 'is-a' reln.ship with base_seq
virtual class base_seq_decorator extends dsi_base_seq;
  rand dsi_base_seq layer;

  rand bit[3:0] num_of_lanes;
  rand int num_of_pkts;
  rand bit hs_rx_mode;
  typedef enum {END_OF_HS, HBP_OR_HFP} lp_mode_substitution;
  rand lp_mode_substitution_e;
  rand bit alp_mode;
  rand bit[27:0] sync_type;

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

  function void set_constr_layer(dsi_base_seq layer);
    this.layer=layer;
  endfunction

  constraint constr_layer {layer.num_of_lanes == num_of_lanes;
                           layer.num_of_pkts  == num_of_pkts;
                           layer.hs_rx_mode == hs_rx_mode;
                           layer.lp_mode_substitution_e == lp_mode_substitution_e;
                           layer.alp_mode == alp_mode;
                           layer.sync_type == sync_type;
                          }
endclass:base_seq_decorator
```

Figure 2. Base Decorator class

3) Concrete decorator class: - This class encapsulates the constraints specific to each of the modes/versions etc. Figure 3 below has code for multiple concrete decorator classes.

```
class cphy_mode1_decorator extends base_seq_decorator;
  `uvm_object_utils (cphy_mode1_decorator)
  function new(string name="cphy_mode1_decorator");
    super.new(name);
  endfunction

  //This constraint is specific to Mode1 in CPHY
  constraint cons_cphy_mode1 {
                       num_of_lanes inside {[1:2]};
                       num_of_pkts  inside {1:128]};
                       lp_mode_substitution_e dist {END_OF_HS:/40, HBP_OR_HFP:/ 60}; }
                      }
endclass:cphy_mode1_decorator

class dphy_mode2_decorator extends base_seq_decorator;
  `uvm_object_utils (dphy_mode2_decorator)
  function new(string name="dphy_mode2_decorator");
    super.new(name);
  endfunction

  //This constraint is specific to Mode2 in DPHY
  constraint cons_dphy_mode2 {
                       num_of_lanes inside {[1:5]};
                       num_of_pkts  inside {1:32]};
                       lp_mode_substitution_e == END_OF_HS;
                      }
endclass:dphy_mode2_decorator
```

```
class v1_1_decorator extends base_seq_decorator;
  `uvm_object_utils (v1_1_decorator)
  function new(string name="v1_1_decorator");
    super.new(name);
  endfunction

  //This constraint is specific to Mode1 in v1.1
  constraint cons_v1_1 {
                        hs_rx_mode == 0;
                        alp_mode ==0 ;
                        sync_type == 28'h3444440}; //Sync Type 0
                       }
endclass:v1_1_decorator

class v1_2_decorator extends base_seq_decorator;
  `uvm_object_utils (v1_2_decorator)
  function new(string name="v1_2_decorator");
    super.new(name);
  endfunction

  //This constraint is specific to Mode2 in v1.2
  constraint cons_v1_2 {
                        hs_rx_mode == 0;
                        alp_mode ==0 ;
                        sync_type inside {28'h3444440,//Sync Type 0
                                          28'h3444441,//Sync Type 1
                                          28'h3444442,//Sync Type 2
                                          28'h3444443,//Sync Type 3
                                          28'h3444444 //Sync Type 4
                                          };
                       }
endclass:v1_2_decorator
```

Figure 3. Concrete decorator classes

Each of the ‘concrete decorator’ classes present in Figure 3 are wrapped around the base sequence. The constraints present in this ‘concrete decorator’ class then get applied to the variables present in the base sequence. In effect, the constraints present in the ‘concrete decorator’ class are layered over the constraints present in the base sequence. One powerful feature of this technique is that any number of the ‘concrete decorator’ classes can be active at time over the course of the simulation. Thus, we can achieve huge number of combinations of constraints specific to all the configurations without needing to have a derived class for each combination.

Figure 1 has code for base sequence (*dsi_base_seq*) in our example scenario. It is wrapped around by the base decorator class which is an abstract class (*base_seq_decorator*) as shown in Figure 2. Each of the ‘concrete decorator’ classes denoting a different configuration/version are shown in Figure 3. All these classes are used in the test case (as shown is Figure 4) to generate constraint random stimulus.

Figure 4 below shows a ‘concrete decorator’ class, *cphy_mode1_decorator*, which encapsulates the constraints for a specific mode of CPHY layer, being applied onto the variables in the base sequence. It is then followed by a *dphy_mode1_decorator* being applied for another instance of the base sequence. As already mentioned above multiple ‘concrete decorator’ classes can be active at a time in the course of a simulation. ‘MULTIPLE LAYER CONSTRAINTS’ section of Figure 4 shows the layering of 2 specific decorators onto the base sequence variables. Thus, we could achieve the stimulus for CPHY specific cases on v1.1 with minimal effort.

```
class dsi_base_test extends uvm_test;

  `uvm_component_utils(dsi_base_test)
  //Base sequence instantiation
  dsi_base_seq dsi_base_seq_inst;
  //Specific decorator class instantiations
  cphy_mode1_decorator constr_cphy_mode1;
  dphy_mode1_decorator constr_dphy_mode1;
  v1_1_decorator constr_v1_1;
  v1_2_decorator constr_v1_2;

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

  function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    dsi_base_seq_inst=dsi_base_seq_inst::type_id::create("dsi_base_seq_inst");
  endfunction : build_phase

  task run_phase(uvm_phase phase);
    //Randomization of the base seq with out any constraints applied
    dsi_base_seq_inst.randomize();

    //*************** SINGLE LAYER CONSTRAINTS*******************
    //Randomization of the base seq with CPHY mode1 specific decorator layered over it.
    //These statements generate the randomized stimulus as per the constraints present
    //in 'cphy_mode1_decorator' specific decoraor class
    constr_cphy_mode1.set_constr_layer(dsi_base_seq_inst);
    constr_cphy_mode1.randomize();
    //Randomization of the base seq with CPHY mode1 specific decorator layered over it.
    //These statements generate the randomized stimulus as per the constraints present
    //in 'dphy_mode1_decorator' specific decoraor class
    constr_dphy_mode1.set_constr_layer(dsi_base_seq_inst);
    constr_dphy_mode1.randomize();

    //*************** MULTI LAYER CONSTRAINTS*******************
    //Randomization of the base seq with 'CPHY mode1 specific decorator' layered over it.
    //These constraints are further layered with 'v1.2 specific decorator' constraints layered over it
    //These statements generate the randomized stimulus as per the constraints present
    //in 'cphy_mode1_decorator' specific decorator class solved together with 'v1.2' constraints
    constr_cphy_mode1.set_constr_layer(dsi_base_seq_inst);
    constr_v1_2.set_constr_layer(dsi_base_seq_inst);
    constr_v1_2.randomize();

    //Randomization of the base seq with 'DPHY mode1 specific decorator' layered over it.
    //These constraints are further layered with 'v1.1 specific decorator' constraints layered over it
    //These statements generate the randomized stimulus as per the constraints present in
    //'dphy_mode1_decorator' specific decorator class solved together with 'v1.1' constraints
    constr_dphy_mode1.set_constr_layer(dsi_base_seq_inst);
    constr_v1_1.set_constr_layer(dsi_base_seq_inst);
    constr_v1_1.randomize();
  endtask : run_phase
endclass: dsi_base_test
```

Figure 4. Test case with decorator pattern used

Figure 5 below gives the UML diagram specific to decorator pattern: -

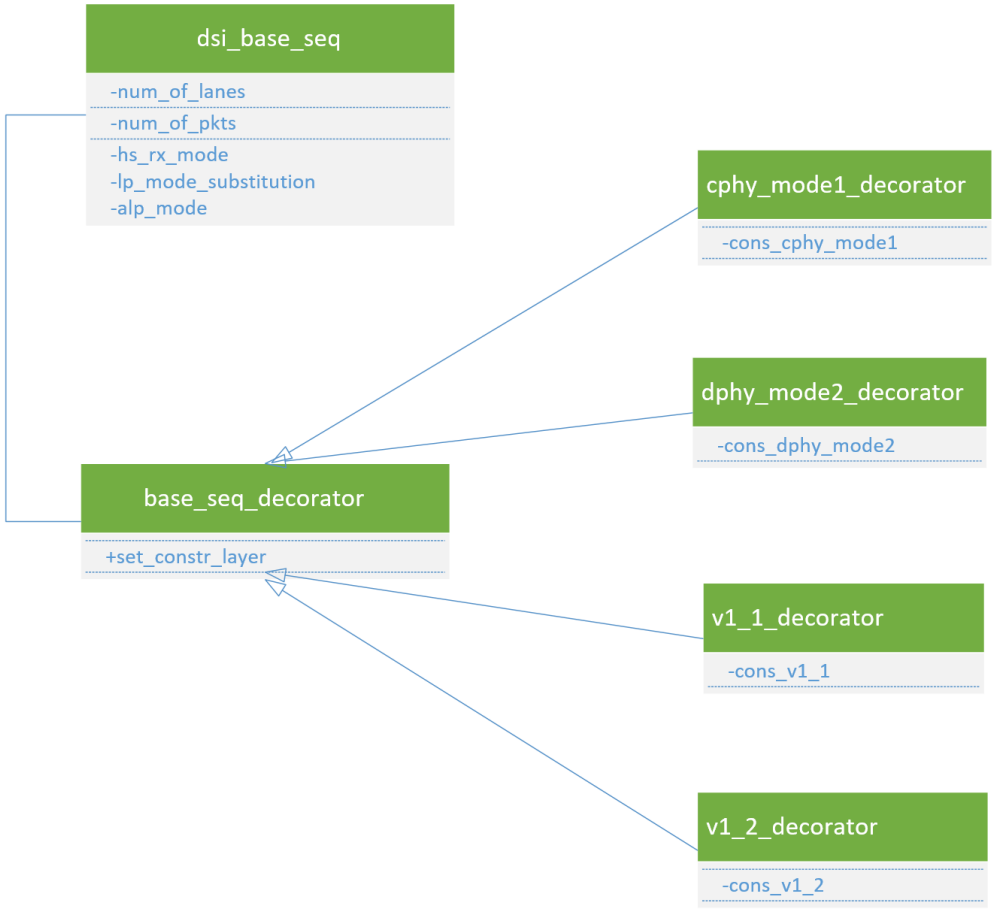

Figure 5. Decorator pattern UML diagram

IV. ACCOMODATING PACKET STRUCTURE DIFFERENCES

One of the common scenarios that we observe in verification is that the packet structure keeps changing across different versions of the specification or across different PHY layers. For DSI protocol, the HS Mode packet structure is different between CPHY and DPHY layers. CPHY has a 12-bit CRC calculated over its Header contents for error detection, while DPHY has an ECC calculated over its header for the same. The fields in the Packet are also different along with their packing behavior. CPHY packet has Sync Symbol Detection Code (SSDC) embedded in its header while DPHY doesn't have such requirement. Figure 6 & 7 below shows the differences in the Packet structure of both CPHY and DPHY. We have tried to explain below how this scenario is tackled generally and how a specific design pattern can be used for this problem to make the code more structured.

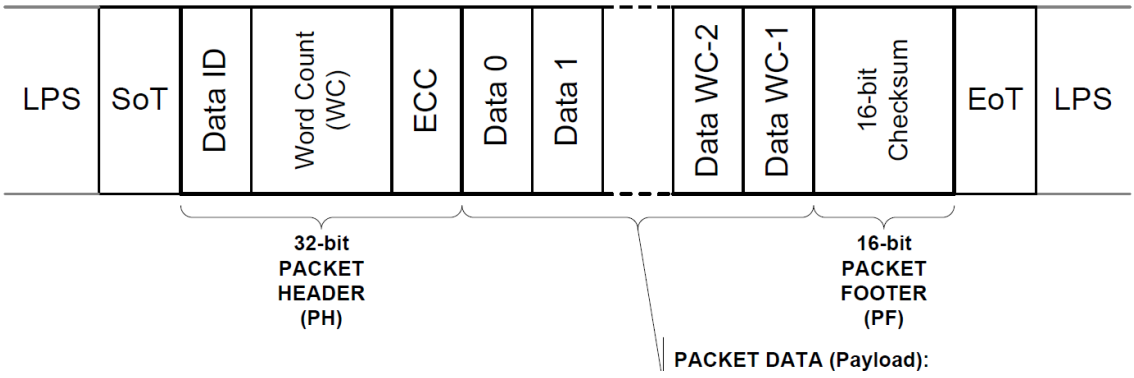

Figure 6. DPHY Long Packet structure

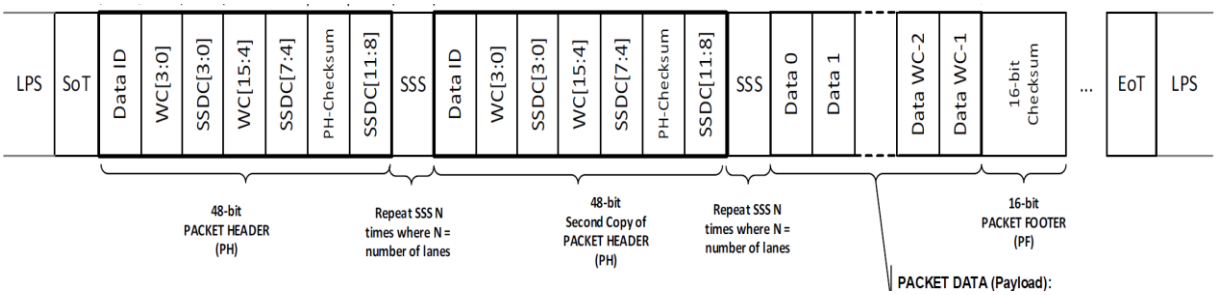

Figure 7. CPHY Long Packet structure

Standard Approach

This scenario is generally handled either by modifying the existing packet transaction to add/remove new fields and conditionally add ECC/CRC check for the Header contents or by creating a derived packet transaction class with all the added functionality. This is a clumsy approach as it either involves modifying the existing code base or creating a huge number of derived classes as the verification environment keeps expanding to support new versions.

Proposed Approach : Use Strategy pattern

Strategy pattern is defined as below as per [2]:

"Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it."

Typically packet transaction classes implement two types of behaviors, 'pack/unpack' to convert from/to bit-level representations and 'check behavior' to check for the errors which would be typically ECC or CRC. These behaviors differ between CPHY and DPHY. The idea is to encapsulate these behaviors and then use the appropriate behavior interchangeably depending on the specific situation. This not only makes code more readable but also more extensible. A changed packet structure for a newer version of specification or a newer PHY layer can easily be encapsulated by adding its own logic as the newer 'pack behavior' or 'check behavior'.

For this approach we use an OOP concept called 'composition' which makes use of the 'interface class' that has been introduced into the system verilog standard from IEEE 1800-2012. Composition is typically used in software world as an alternative to inheritance. In composition, the required functionality is achieved by assembling the behaviors from various objects. Objects that are 'composed' should have well-defined interfaces, for which, we make use of the 'interface classes'. We define an interface class for each of the behaviors that the Packet transaction class should model i.e., 'Pack behavior', 'Unpack behavior' & 'Check behavior'. These functions must be provided when an interface class is implemented. Figure 8 shows the code for the interface classes encapsulating each of the behaviors.

```
// Fields class
class pkt_fields extends uvm_object;
  rand bit[1:0] vc_id;
  rand bit[5:0] data_id;
  rand bit[15:0] word_count;
  rand bit[7:0] payload[];
endclass: pkt_fields

// Pack/UnPack behavior interface class
interface class pack_unpack_behavior #(type pkt_fields = fields);
  pure virtual function void pack();
  pure virtual function void unpack();
endclass: pack_unpack_behavior

// Pack/UnPack behavior interface class
interface class check_behavior #(type pkt_fields = fields);
  pure virtual function check();
endclass: check_behavior
```

Figure 8. Interface class definitions

Figure 9 below shows the code for the specific behavior implementation of each of the interface classes.

```
//32 bit Pack/Unpack behavior implementation
class 32bit_pack_unpack implements pack_unpack_behavior;
  bit[31:0] raw_data;
  virtual function fields unpack(logic [31:0] raw_data)
     fields fields_inst =new();
     fields_int.vc_id = raw_data[1:0];
     fields_int.data_id = raw_data[7:2];
     fields_int.word_count = raw_data[23:8];
     //add rest of the pack behavior as per DPHY Pkt structure
     return  fields_inst;
  endfunction

  virtual function bit[31:0] pack(pkt_fields fields_inst)
     fields_inst =new();
     raw_data[1:0] = fields_inst.vc_id;
     raw_data[7:2] = fields_inst.data_id;
     raw_data[23:8] = fields_inst.word_count;
     //add rest of the unpack behavior as per DPHY Pkt structure
     return raw_data;
  endfunction
endclass: 32bit_pack_unpack

//48 bit Pack/Unpack behavior implementation
class 48bit_pack_unpack implements pack_unpack_behavior;
  virtual function pkt_fields unpack(logic [47:0] raw_data)
     fields fields_inst =new();
     //add behavior as per CPHY Pkt structure
  endfunction

  virtual function bit[47:0] pack(pkt_fields fields_inst)
     fields_inst =new();
     //add behavior as per CPHY Pkt structure
  endfunction
endclass: 48bit_pack_unpack

class ecc implements check_behavior;
  virtual function bit[7:0] check(pkt_fields fields_inst)
     //add logic specific to ecc here
  endfunction
endclass: ecc

class crc implements check_behavior;
  virtual function bit[7:0] check(pkt_fields fields_inst)
     //add logic specific to ecc here
  endfunction
endclass: crc
```

Figure 9. Interface class implementations

Figure 10 below shows the code for the base packet class and the DPHY and CPHY packet classes which are composed depending on their specific behaviors

```
virtual class dsi_packet extends uvm_sequence_item;
  pkt_fields fields_inst= new();
  pack_unpack_behavior pack_unpack_behavior_inst;
  check_behavior check_behavior_inst;

  //This function sets the Pack/Unpack behavior for the specific pkt format
  function set_pack_unpack_behavior(pack_unpack_behavior pack_unpack_behavior_inst);
     this.pack_unpack_behavior_inst=pack_unpack_behavior_inst;
  endfunction
  //This function sets the Check behavior for the specific pkt format
  function set_check_behavior(check_behavior check_behavior_inst);
     this.check_behavior_inst=check_behavior_inst;
  endfunction
  //This function performs the 'Pack' operation for the specific pkt format
  function perform_pack();
     pack_unpack_behavior_inst.pack();
  endfunction
  //This function performs the 'Unpack' operation for the specific pkt format
  function perform_unpack();
     pack_unpack_behavior_inst.unpack();
  endfunction
  //This function performs the 'check' operation for the specific pkt format
  function perform_check();
     check_behavior_inst.check();
  endfunction
endclass: dsi_packet

//DPHY Pkt class
class dsi_dphy_packet extends dsi_packet;
  32bit_pack_unpack 32_bit_pack_unpack_inst;
  ecc ecc_check_inst;

  function new();
    //Required behaviors for the DPHY Pkt format
    32_bit_pack_unpack_behavior_inst=new();
    ecc_check_behavior_inst=new();
    //Set those required behaviors
    set_pack_unpack_behavior(32_bit_pack_unpack_behavior_inst);
    set_check_behavior(ecc_check_inst);
  endfunction
endclass: dsi_dphy_packet


//CPHY Pkt class
class dsi_cphy_packet extends dsi_packet;
  48bit_pack_unpack 32_bit_pack_unpack_inst;
  ecc ecc_check_inst;

  function new();
    //Required behaviors for the CPHY Pkt format
    48_bit_pack_unpack_behavior_inst=new();
    ecc_check_behavior_inst=new();
    //Set those required behaviors
    set_pack_unpack_behavior(48_bit_pack_unpack_behavior_inst);
    set_check_behavior(ecc_check_inst);
  endfunction
endclass: dsi_cphy_packet
```

Figure 10. DPHY & CPHY packets composed from their specific behaviors

As an when a new packet structure needs to be added or supported the ‘behaviors’ specific to those packet structures can be implemented and ‘composed’ into a newer packet. Figure 11 below shows code for the newer version of the specification has ‘packing behavior’ specific to CPHY but the ‘checking behavior’ specific to DPHY and using this pattern we can ‘compose’ the packet appropriately. This would not be possible with inheritance as ‘multiple inheritance’ is not supported in system Verilog.

```
//Newer Pkt class
class dsi_new_packet extends dsi_packet;
  48bit_pack_unpack 48_bit_pack_unpack_inst;
  ecc ecc_check_inst;

  function new();
    //Required behaviors for the Newer Pkt format
    48_bit_pack_unpack_behavior_inst=new();
    ecc_check_behavior_inst=new();
    //Set those required behaviors
    set_pack_unpack_behavior(48_bit_pack_unpack_behavior_inst);
    set_check_behavior(ecc_check_inst);
  endfunction
endclass: dsi_new_packet
```

Figure 11 Packet structure for new packet

The UML diagram for the Packet transaction class explained above is given in Figure 12 below.

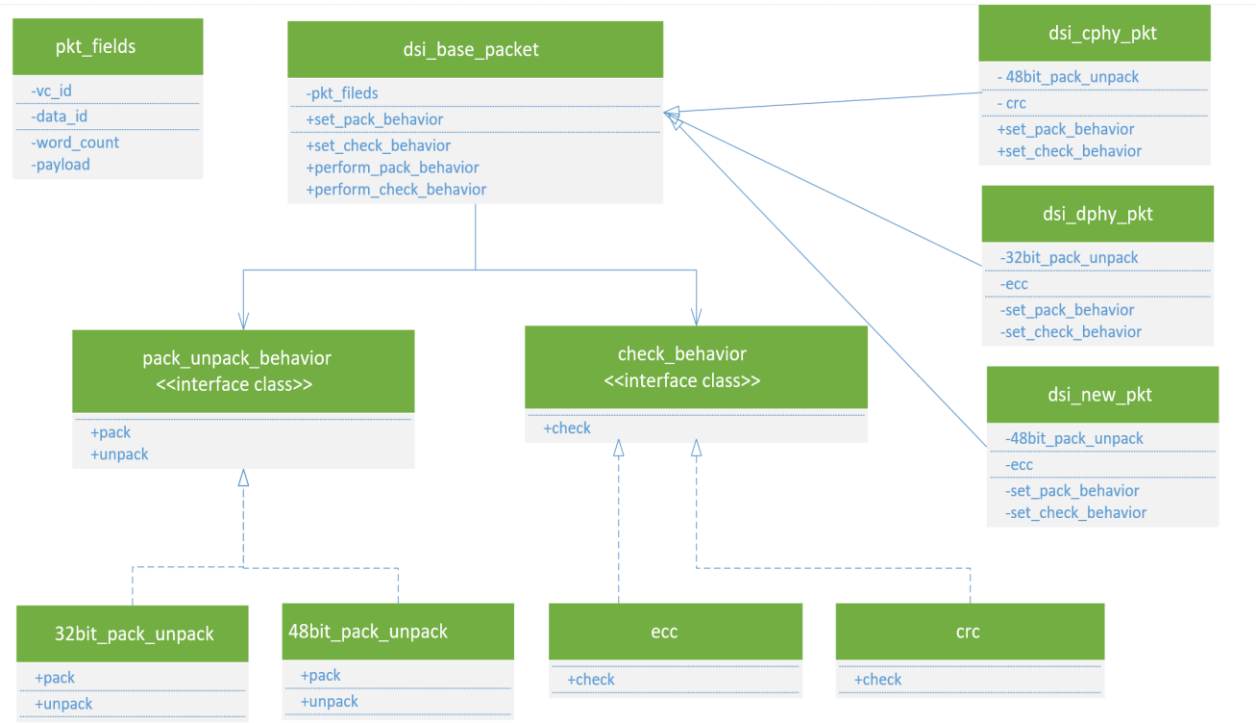

Figure 12: UML diagram for Strategy pattern

V. ADDING NEW CODE TO THE EXISTING CLASS STRUCTURE

As the size of the testbench grows and as we need to support the newer configurations and versions of the specification, we may need to add new print messages or perform certain configuration checks deep within the class hierarchy to aid in debugging. The DSI testbench made use of various third-party VIPs which had to be configured into multiple configurations depending on the mode that is being verified. As the number of configurations and versions to be supported increased we had to check the configuration of various IPs in various modes. Also, we had to add print messages to aid us in debug. We explained below as to how this is accomplished generally and how using visitor pattern accomplishes this in a better way.

Standard Approach

A standard way is to add the required code by modifying each of the classes in the verification environment that needs the new functionality to be added to them. This may require us to modify the third-party VIPs that are

generally used by multiple teams within the company and hence is hugely inefficient. This also violates one of the most important solid design principles called the 'open closed principle', which mentions that "Software entities (classes, functions etc. should be open for extension but closed for modification. Another way to achieve this would be to create a derived class for each of the classes that we wanted to modify and then override the derived class with the base class throughout the testbench. This is a very inefficient approach as it make debug very difficult. Also, some of the VIPs we used were the vendor VIPs and hence it was not even possible to add the code as they are encrypted.

Proposed Approach : Use Visitor pattern

Reference [2] defines Visitor pattern as below:

"Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates."

This pattern is suitable to an environment with a well-defined class structure, which, a UVM based environment is. Typically, implementation of Visitor pattern should be planned beforehand to make sure that all the classes in testbench are compliant with the visitor flow. But from UVM 1.2, UVM library provides all the necessary infrastructure needed to implement this pattern. Any class that is extended indirectly or directly from the 'uvm_component' class will be able to handle the 'visitor' access.

Below are the 2 main components of the visitor infrastructure provided by the UVM library:

1) uvm_visitor: - This is an abstract class which needs to be extended and then is added with the new functionality that we want to add to the specific component that we wish to modify. This extended class is called a 'concrete visitor' class. This uvm_visitor class also contains the pre-processing hook function 'begin_v' and post-processing hook function 'end_v' which can be implemented in the 'concrete visitor' class to be able to initialize or observe any activity in the classes that would be 'visited'.

2) uvm_visitor_adapter: - This is an abstract class that wraps around the components that need to be 'visited' to be added with the new functionality that we implement in concrete visitor class. A variety of adapter classes like *uvm_top_down_vistor_adatper* (top-down), *uvm_bottom_up_visitor_adapter* (bottom-up) , *uvm_by_level_adapter* (level-by-level) are defined in the UVM library to traverse the class structure in a particular way. UVM library also provides an abstract class *uvm_structure_proxy* that provides all the sub elements in the structure of a certain element. *uvm_component_proxy* is a specialized class of this abstract class that provides all the subcomponents of a specific uvm_component.

Adding display messages to aid in debug

When our existing CPHY testbench had to be extended to support DPHY layer, we had a requirement to print the transactions that are being pushed into a queue in the scoreboard and had to keep tract of size of the queue. Given below in Figure 13, is the code for the 'visitor' class that encapsulates the functionality that we wish to add which in our case is the ability to print transactions.

```
//Visitor Class encapsulating the functionality that is to be newly added.
class dsi_dphy_sb_print_vistor extends uvm_vistor;
  function new (string name=" ");
    super.new(name);
  endfunction:new

  virtual function void visit (uvm_component node);
    if (node.get_object_type() == dsi_dphy_visitor_sb::type_id::get()) begin
      fork
        //The following function is called on each component that this class visits
        dsi_dphy_print_visit(node);
      joine_none
    end
  endfunction:visit
  //The following function prints the Actual Txns in queue in the scoreboard
  //It also prints the size of the queue
  virtual task dsi_dphy_print_visit (uvm_component node);
    dsi_dphy_visitor_sb dsi_sb;
    $cast(sb,node);

    while(1) begin
      @(dsi_sb.actual_tx_que.size())
      `uvm_info(get_type_name(), $psprintf("Actual txn at serial i/f is  = %s ", actual_tx_que.convert2string()), UVM_HIGH)
      `uvm_info(get_type_name(), $psprintf("Size of the Actual serial txn is updated. It's value is  =%d", dsi_sb.actual_tx_que.size()),UVM_HIGH)
    end
  end
  endfunction:dsi_dphy_print_visit
endclass:dsi_dphy_sb_print_vistor
```

Figure 13. Visitor class with the scoreboard ‘print’ messages added

Given below in Figure 14, is the code for adapter class that wraps the scoreboard and applies the functionality that is implemented in visitor class.

```
//Adapter class that 'traverses' the components
class dsi_dphy_adapter extends uvm_visitor_adapter;
  function new (string name=" ");
    super.new(name);
  endfunction
  //This function applies the 'visit' function on all the compoenents 'visited'
  virtual function void accept (uvm_component s, uvm_visitor v, uvm_structure_proxy#(uvm_component) -, bit invoke_begin_end=1);
    if(invoke_begin_end)
       v.begin_v();

       v.visit(s);

    if(invoke_begin_end)
       v.end_v();
  endfunction
endclass:dsi_dphy_adapter
```

Figure 14. Adapter class that traverses the specified components

Given below in figure 16 is the code that needs to be added to the run phase of the environment class so that the print statements present in the visitor class are added conditionally to the scoreboard.

```
//This code has to be added in the environment class of the testbench
class dsi_env extends uvm_env;
  //All the code specific to other funcionality of env class of the testbench

  task run_phase (uvm_phase phase);
    // Add all the code specfic other functionality

    //-----
    //-----

    // Add all the code specfic to visitor functionality
    dsi_dphy_sb_print_vistor dphy_sb_vistor_inst;
    dsi_dphy_adapter adapter_inst;

    dphy_sb_vistor_inst = new("dphy_sb_vistor_inst");
    adapter_inst = new("adapter_inst");

    //Give the instance of the scoreboard as the first arguement  to accept function call below
    adapter_inst.accept(dsi_sb_inst,dphy_sb_vistor_inst,null);
 endtask
endclass:dsi_env
```

Figure 15. Environment class with the ‘visitor’ traversal logic added

Checking the configurations info. for various components

As the testbench is expanded to support multiple PHY layers and versions, the components in the test bench (VIPs etc.) are configured in various modes. It becomes important to check if the configuration is properly set as per the newly added modes before proceeding further into simulation. Given below in Figure 16, is the code for the 'visitor' class that encapsulates the functionality where we try to check the configuration information of various components at the beginning of the simulations. This can be extended to periodic check depending on the debug needs.

```
//Visitor Class encapsulating the functionality that is to be newly added. Configuration check in this case
class dsi_cfg_check_vistor extends uvm_vistor;
  function new (string name=" ");
    super.new(name);
  endfunction:new

  virtual function void visit (uvm_component node);
    //Check for a specific variable being set in the VIP agent cfg and for rest if the components just print the cfg
    if (node.get_object_type() == visitor_dsi_vip_agent::type_id::get()) begin
        visit_vip_agent(node);
    end else
      `uvm_info(get_type_name(), $psprintf("cfg of component %s is %s", node.get_full_name(),cfg.convert2string()),UVM_HIGH)
    end
  endfunction:visit

  //If the component being visited is VIP agent perfrom a specific functionality
  virtual function void visit_vip_agent(uvm_component node);
    visitor_dsi_vip_agent dsi_vip_agent;
    $cast(dsi_vip_agent,node);

    if(dsi_vip_agent.cfg.phy_mode == `CPHY && dsi_vip_agent.cfg.phy_mode == `V1_2) begin
      `uvm_info(get_type_name(), $psprintf("CFG is set to CPHY and Version V1.2", UVM_HIGH)
      //Perform some functionality that is specific to CPHY V1.2
    end else if(dsi_vip_agent.cfg.phy_mode == `DPHY && dsi_vip_agent.cfg.phy_mode == `V1_1) begin
      `uvm_info(get_type_name(), $psprintf("CFG is set to DPHY and Version V1.1", UVM_HIGH)
      //Perform some functionality that is specific to DPHY V1.1
    end
  endfunction:visit_vip_agent
endclass:dsi_cfg_check_vistor
```

Figure 16. Visitor class encapsulating the 'cfg check' functionality

Given in Figure 17 below, is the code for the *uvm_top_down_adapter_class* that wraps the environment. Using *uvm_component_proxy* instantiation we can traverse all the components that are subcomponents of the environment class and perform operations on each of them. In this case we are printing the configuration object, cfg, of each component and if the component is VIP agent, we are performing some additional check.

```
//configuration needs to be checked for all components in hierarchy and hence
// we need to use 'uvm_top_down_adapter'
class dsi_cfg_adapter extends uvm_top_down_adapter;
  function new (string name=" ");
    super.new(name);
  endfunction

  virtual function void accept (uvm_component s, uvm_visitor v, uvm_structure_proxy#(uvm_component) -, bit invoke_begin_end=1);
    if(invoke_begin_end)
       v.begin_v();

       v.visit(s);

    if(invoke_begin_end)
       v.end_v();
  endfunction
endclass:dsi_cfg_adapter
```

Figure 17. Top down adapter to traverse all components in the hierarchy

Below code in Figure 18 needs to be added in the run phase of the environment class so that the required functionality is achieved.

```
//This code has to be added in the environment class of the testbench
class dsi_env extends uvm_env;

  //All the code specific to other funcionality of env class of the testbench

  task run_phase (uvm_phase phase);
    // Add all the code specfic other functionality

    //-----
    //-----

    // Add all the code specfic to visitor functionality
    dsi_cfg_check_vistor vip_cfg_check_vistor_inst;
    dsi_cfg_adapter adapter_inst;
    uvm_component_proxy proxy;

    dphy_sb_vistor_inst = new("dphy_sb_vistor_inst");
    adapter_inst = new("adapter_inst");
    proxy = new("proxy");

    //Give the instance of the scoreboard as the first arguement below
    adapter_inst.accept(this,vip_cfg_check_vistor_inst,proxy);
 endtask
endclass:dsi_env
```

Figure 18. Environment class that has all its subcomponents traversed by visitor logic

Figure 19 below gives the UML diagram for the both the above scenarios implemented in visitor pattern.

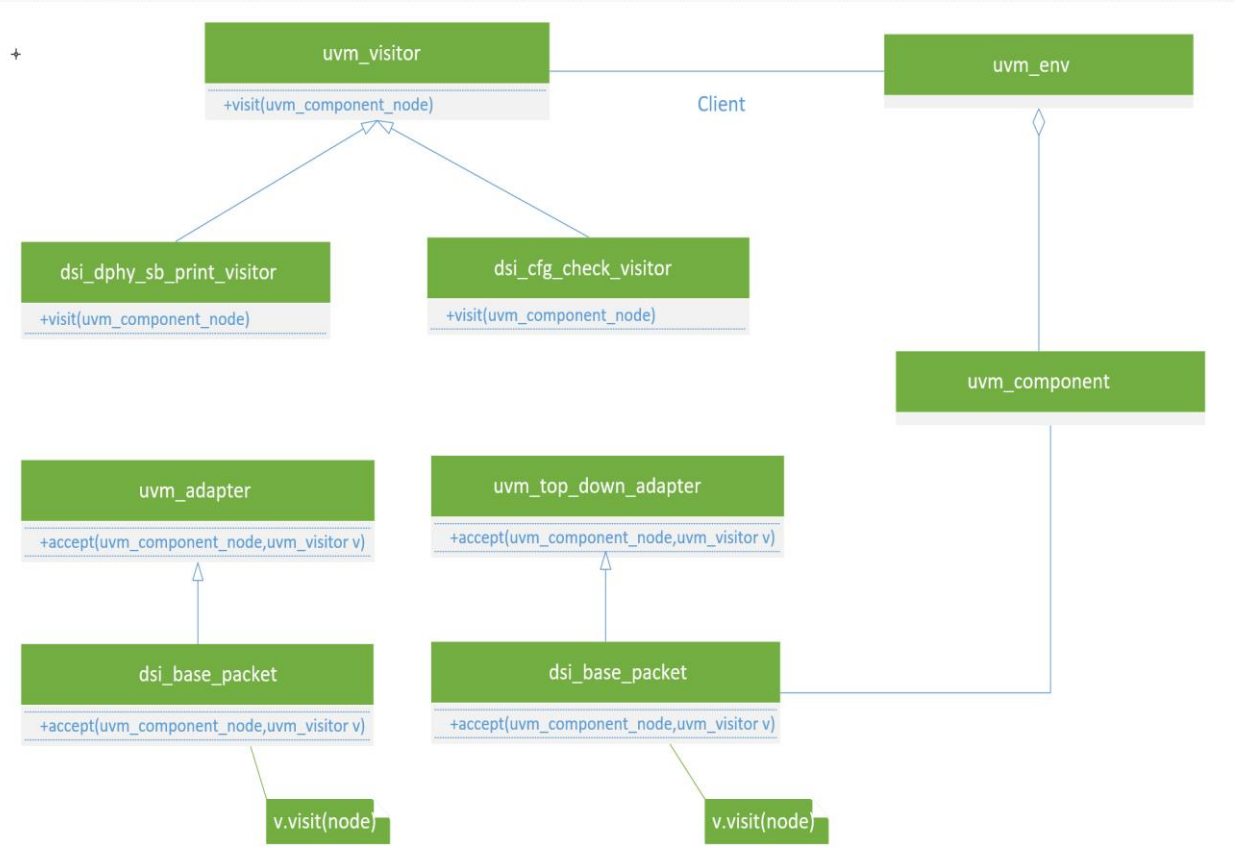

Figure 19. UML diagram for visitor pattern

VI. DYNAMICALLY SHIFTING BETWEEN MULTIPLE MODES

DSI is a highly configurable IP. As we keep extending the testbench to support multiple modes of operation the test case coding becomes complex as we need to test the design for dynamic shifting between multiple modes. A few such modes supported are High speed mode (HS), Low power Mode (LP) &Ultra Low power

state (ULPS). To be able to test this behavior DUT must be configured to operate in each of these modes and the VIPs used in the testbench also need to be configured correspondingly. This must be done multiple times throughout the test case. A standard way generally followed to achieve this is explained below along with the specific design pattern to achieve this better way.

Standard Approach

Each time we need to change the mode, the state of the DUT and the testbench (comprising a bunch of configuration registers and config objects) needs to be updated. This may be done by specifically modifying the config objects each time as we shift dynamically from one mode to the next. This is a tedious approach as it involves code duplication and may be buggy if not done carefully.

Proposed Approach : Use Memento pattern

Memento is a behavioral design pattern and is suitable when we want to save the state of an object so that it can be restored later. As the complexity of the test case increases, we may need to save certain 'check points' (referring to specific modes in our case) of the state of the DUT and test bench so that we can revisit them later to be able to shift the modes dynamically with minimal effort.

This pattern consists of 3 main components: -

1) Memento: - This is the class that encapsulates the content that needs to be stored so that it can be restored later dynamically.

2) Originator:- This class creates the object of the memento class and then saves its present state. It also makes use of the previously used memento states to be restored later if required.

3) Caretaker:- This class keeps track of all the saved states in the originator and can request for a specific state that needs to be restored.

Given below in Figure 20, is the Memento class that encapsulates the 'configuration' instance(cfg) that needs to be stored: -

```
//Memento class that contains the information that we need to be stored
class dsi_memento extends uvm_object;
  `uvm_object_utils (dsi_memento)
  function new (string name="dsi_memento");
    super.new(name);
  endfunction:new

  // Instance of the configuration to be stored
  dsi_config cfg = new();

  // Set the configuration passed the local varaible
  function set_config (dsi_config cfg);
    this.cfg = cfg;
  endfunction:set_config

  // get the configuration stored in the local varaible
  function dsi_config get_config ();
    retrun this.cfg;
  endfunction:get_config
endclass:dsi_memento
```

Figure 20. Memento class containing the 'content' to be saved

Given below in Figure 21 is the originator class which instantiates a memento class and saves the configuration information of the current mode.

```
//Originator class that stores various 'states' of cfg
class dsi_originator extends uvm_object;
  `uvm_object_utils (dsi_originator)
  function new (string name="dsi_originator");
    super.new(name);
  endfunction:new

  dsi_config cfg;

  // Set the present state of 'cfg' to local variable
  function set_state (dsi_config cfg);
    this.cfg = cfg;
  endfunction:set_config

  // Get the present state of 'cfg' stored in the local variable
  function dsi_config get_state ();
    return this.cfg;
  endfunction:get_config

  // Save the  the present state of 'cfg' to memento
  function dsi_memento store_cfg_state_in_memento ();
    dsi_memento dsi_memnto_inst;
    dsi_memento_inst=dsi_memento::type_id::create("dsi_memento_inst");
    dsi_memento_inst.set_config(cfg);
    return dsi_memento_inst;
  endfunction:store_cfg_state_in_memento

  // Retreive the the present state of 'cfg' from memento
  function void restore_cfg_state_from_memento ();
     cfg = dsi_memento_inst.get_config();
  endfunction:store_cfg_state_in_memento
endclass:dsi_originator
```

Figure 21. Originator class that saves/retrieves the present state

Given below in Figure 22 is the caretaker class that keeps track of all configurations of all modes and then requests the specific value when needed.

```
//This class requests the specifc state of memento to be restored
class dsi_caretaker extends uvm_object;
  `uvm_object_utils (dsi_caretaker)
  function new (string name="dsi_caretaker");
    super.new(name);
  endfunction:new

  //Queue of the memento classes to save multiple states
  dsi_memento memento_que[$];

  //Add a new state to the memnto queue
  function add_state (dsi_memento cfg_state);
    memento_que.push_back(cfg_state);
  endfunction:set_config

  //Retreive a new state from the memnto queue
  function dsi_memento retreive_state (int idx);
    return memento_que[idx];
  endfunction:get_config
endclass:dsi_caretaker
```

Figure 22. Caretaker class storing and retrieving multiple states

Given in the Figure 23 below is the test case class code where we try to dynamically shift between multiple modes.

```
//This code has to be added in the specific test case
class dsi_test extends uvm_test;

  //All the code specific to other funcionality of test case

  task run_phase (uvm_phase phase);
    //Enter HS mode.
    dsi_originaror.set_state(hs_cfg);

    //Perfrom all the HS mode specific functionality of the test case

    //Shift to LP mode. Store the HS mode configuration in the memento queue before entering LP mode
    dsi_caretaker.add(dsi_originaror.store_cfg_state_in_memento());
    //Enter LP mode.
    dsi_originaror.set_state(lp_cfg);

    //Perfrom all the LP mode specific functionality of the test case

    //Exit LP mode.
    dsi_originator.restore_cfg_state_from_memento(dsi_originaror.retreive_state());

    //HS Mode state entered again

    //Perfrom all the HS mode specific functionality of the test case

    //Shift to ULPS mode. Store the HS mode configuration in the memento queue before entering ULPS mode
    dsi_caretaker.add(dsi_originaror.store_cfg_state_in_memento());
    //Enter ULPS mode.
    dsi_originaror.set_state(ulps_cfg);

    //Perfrom all the ULPS mode specific functionality of the test case

    //Exit ULPS mode.
    dsi_originator.restore_cfg_state_from_memento(dsi_originaror.retreive_state());

    //HS Mode state entered again

    uvm_component_proxy proxy;

    dphy_sb_vistor_inst = new("dphy_sb_vistor_inst");
    adapter_inst = new("adapter_inst");
    proxy = new("proxy");

    //Give the instance of the scoreboard as the first arguement below
    adapter_inst.accept(this,vip_cfg_check_vistor_inst,proxy);
 endtask
endclass:dsi_test
```

Figure 23. Test case that dynamically shifts modes using memento pattern

Given below is the UML diagram for the Memento pattern in Figure 24: -

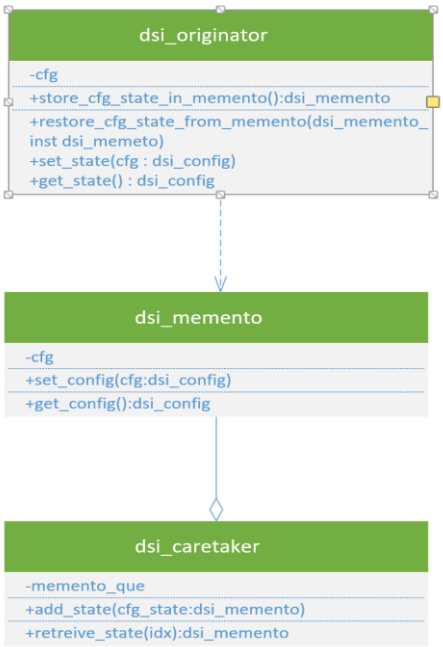

Figure 24. UML diagram for Memento pattern

VII. MODELING TESTBENCH TIMEOUT

As we keep adding support for newer PHY layers and versions, we will keep extending older code to add new functionality. But if some functionalities are already present in the existing code base, we may not need to duplicate it in the newly added code. One such functionality is interface timeout. We need only one instance of the timeout logic per each test. As the code base increases it becomes difficult to track various instances of the timeout logic in the code and this makes the debug process as bit difficult.

Proposed Approach : Use singleton pattern

Singleton pattern suggests that we make sure that all components of the testbench access a single instance of a class that encapsulates the timeout logic. *uvm_event_pool* is one such example where single pattern is used in UVM library. This pattern ensures that a class has only one instance and provides a global point of access to it. A singleton class should have its constructor method defines as protected. The single object that is created of the singleton class is accessed through a static method called *get_timeout_instance()* as shown in below figure 25 below.

```
//Time out logic for the testbench
class dsi_all_intf_timeout ;
  //Add all the clock specific logic here
  real dphy_clk_period, cphy_clk_period;
  logic dphy_clk=0;cphy_clk=0;

  //Add clock generation logic here
  //---
  //---

  dphy_max_no_clock_cycles=2000;//default value
  cphy_max_no_clock_cycles=3000;//default value

  protected function new ();
  endfunction:new

  //Instance handle for this class
  static dsi_all_intf_timeout timeout_inst;

  //Static function to get this class instance
  static function dsi_all_intf_timeout get_timeout_instance();
    if (timeout_inst == null)
    timeout_inst = new();
    retrun this.timeout_inst;
  endfunction:get_timeout_instance

  //timeout logic
  task timeout(bit mode=`DPHY)
    if(mode=`CPHY) begin
      repeat (cphy_max_no_clock_cycles)
       @(posedge (cphy_clk_period);
      end
    else begin
      repeat (dphy_max_no_clock_cycles)
       @(posedge (dphy_clk_period);
      end
    end
  endtask:timeout
endclass:dsi_all_intf_timeout
```

Figure 25. Timeout singleton class

Figure 25 shows the 'timeout' functionality for each mode being encapsulated in a singleton class. Only a single instance of this class can be created, and the same instance needs to be used by any component using it. This aids in debug as it prevents multiple instances being created for the same functionality.

Figure 26 shows the testcase where the singleton timeout class is instantiated, and the timeout value is set.

```
//This code has to be added in the specific test case
class dsi_dphy_test extends uvm_test;

  //All the code specific to other funcionality of teast case

  //Create timeout singleton class instance
  dsi_all_intf_timeout test_timeout ;
  test_timeout dsi_all_intf_timeout::get_timeout_instance();

  task run_phase (uvm_phase phase);
    //Check DPHY specific functionality
    fork
      begin
        dphy_seq.start(dsi_env.dsi_agt.dsi_seqr)
      end
      begin
        test_timeout.timeout(`DPHY);
      end
    join
 endtask
endclass:dsi_dphy_test
```

Figure 26. Test case instantiating timeout singleton class

Figure 27 below gives the UML diagram for the singleton pattern.

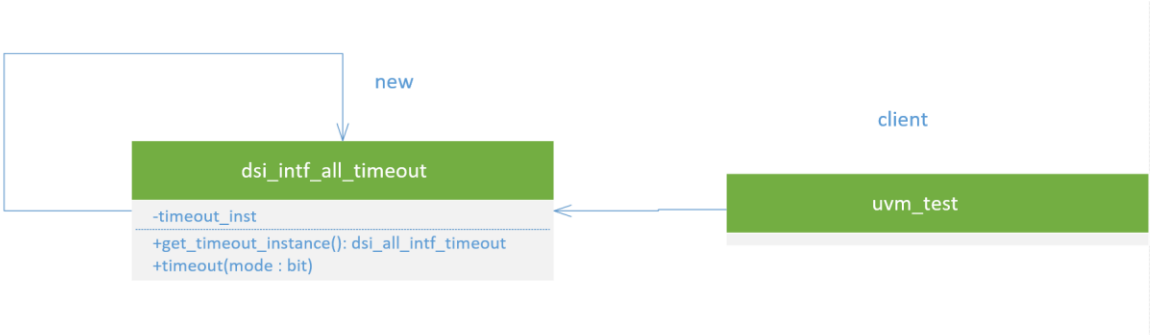

Figure 27. UML diagram for singleton pattern

VIII. CONCLUSION

This paper tries to cover a few of the various design patterns that are being widely used in software world. These are optimized, proven & re-usable solutions to commonly occurring problems in OOPs based coding environments and have evolved over time. Since we try to solve these problems with well-defined patterns that are accompanied with class UML diagrams, the code becomes more readable and extensible. As the complexity of testbenches keeps increasing, such standard well-defined coding practices are needed to solve complex scenarios so that the solutions are scalable for future enhancements. Using these techniques in our testbench helped us in achieving considerable reduction in testbench development time when we had to extend our basic existing testbench, that was coded for a single configuration, to support multiple versions and PHY layers. The test bench was also more readable as each of these patterns was documented with clear UML diagrams hence making it easy for newer team members to understand the testbench. Using these techniques almost cut 20-30% development time that we had to spend on extending the basic testbench supporting one PHY layer for various configurations, by limiting the number of files to that we had to modify to very less number.

IX. FUTURE WORK

Reference [2] defines many design patterns out of which we have tried to make use of a few in our testbench. There are many other patterns like ‘iterator pattern’ and ‘chain of responsibility’ that can find use in complex scoreboarding etc. Each of the design patterns can be further explored to be used efficiently in complex modeling that is needed for in today’s complicated testbenches.

REFERENCES

[1] https://blogs.mentor.com/verificationhorizons/blog/2019/01/29/part-8-the-2018-wilson-research-group-functional-verification-study

[2] E.Gamma, R.Helm, R.Johnson and J.Vlissides, Design Patters: Elements of Re-usable Object-Oriented Software, Indianapolis, IN: Addison-Wesley, 1994.

[3] Ed Nelson, “Design Patterns by Example for SystemVerilog Verification Environments Enabled by SystemVerilog 1800-2102”, DVCON US -2016.

[4] Darko M.Tomusilovic, “Extending functionality of UVM components by using Visitor design pattern” DVCON Europe 2018.

[5] D. M.Tomusilovic, H.J. Arbel “UVM Verification Environment based on Software design patterns ” DVCON US 2018.

[6] https://mipi.org/specs/dsi

[7] Eric Freeman & Elisabeth Freeman with Kathy Sierra & Bert Bates, Head First Design Patterns.