UVM Verification Environment Based on Software Design Patterns

D. M. Tomušilović
Vtool LTD.
Rajićeva 1
Belgrade, Serbia
darkot@thevtool.com
+381654662859

H. J. Arbel
Vtool LTD.
Rajićeva 1
Belgrade, Serbia
hagai@thevtool.com
+972544284574

***Abstract*-Software design pattern is a technique utilized to tackle a commonly occurring problem in the software development. As a significant part of work done by verification engineers includes coding in an object-oriented language, such as SystemVerilog, many of the encountered challenges are suitable to be resolved applying certain design patterns. Their incorporation into the code provides many benefits, contributing to the code reusability and maintainability, and therefore improving the overall code quality. Several practical examples appropriate for employing design patterns are presented in the paper.**

I. Introduction

The paper shows how several design patterns [1] are introduced to a verification environment, to model features implemented within the verified designed logic. The verification environment is developed using the implementation of UVM methodology in SystemVerilog language. As apart from comprising functional verification dedicated elements, such as assertions and covergroups, SystemVerilog supports common object-oriented constructs, it is suitable for adoption of design patterns, a technique utilized in the process of software development. Design patterns define a set of classes and relations between them, which interact in order to resolve a common problem faced during software development. They target system creation, structure and its behavior. The paper presents Memento, Chain of responsibility, Decorator, Strategy, Singleton, Template method and Iterator. Other design patterns are briefly introduced using the practical examples from the project, including Factory, Observer, State, Mediator and Visitor. Some of them are already well-established within the UVM library codebase. Applying design patterns improves code readability, reusability and maintainability, which are requirements typically posed to verification engineers, in order to reduce development cycle time. They are aimed to significantly improve the environment's efficiency, accelerate coding and facilitate code documentation.

II. Design Patterns in the Verification Environment

The verified designed logic contains features whose modelling within the verification environment can be successfully achieved using design patterns. For each of them, the paper provides a problem description, the reason why the problem is suitable to be modelled using a certain design pattern, the UML class diagram [2] along with implemented SystemVerilog codebase in the Appendix, and finally the benefits and drawbacks of the solution.

A. Memento

Memento is a behavioral software design pattern used to capture the state of an object and restore the object into its previous state. In the software development context, it provides the undo mechanism in the applications, promoting the data hiding and not violating the encapsulation principle. In the verification environment, it is utilized to model the "restore" feature. The device is implemented using multiple power domains – PD1 and PD2, as shown in Figure 1. While PD2 is in operational Run Mode (RM), the block configuration is defined in the appropriate register. When the PD2 enters the low-power mode (LPM), the configuration register behaves as read-only and reading the register results in read value isolation. During the LPM, the RM configuration is stored within always-on PD1. Upon the PD2 LPM exit, the content of the configuration register is restored to the value before the LPM entry. Memento is a good candidate to model the content reversal.

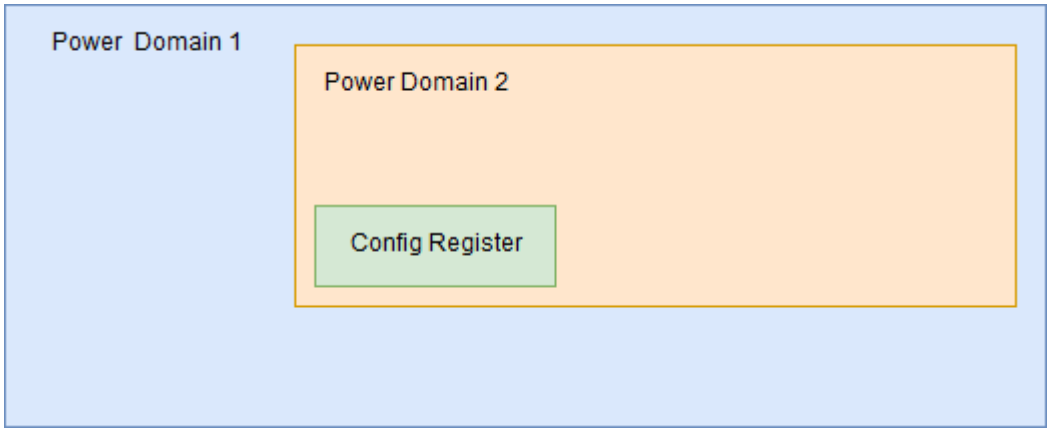

Figure 1. Device Power domains block diagram

The Memento state design pattern defines three main components:

- Memento – Represents a container class for the content to be saved and restored. In the example shown in Figure Appendix-1, Memento class contains a single configuration register, with corresponding get/set methods.
- Originator – Uses the Memento class to save the current state it is in, as shown in Figure Appendix-2.
- Caretaker – Requests from the Originator to save its state. It is aware of all saved states and can request a certain state to be restored, as shown in Figure Appendix-3.

The save and restore operations are demonstrated in Figure 2, with the corresponding log output in Figure 3. Run Mode and LPM register states are captured within the separate objects. During the operational Run Mode, the Run Mode configuration object is set in the Originator. Displaying the current configuration, shows the value of 0x5a5a, configured during RM. Upon the LPM entry, the Caretaker adds the configuration object to the list of saved states it is aware of. The reference in the Originator is updated to point to the object that reflects the LPM register state, with the value of 0. Upon the LPM exit, the Run Mode configuration, being saved by the Caretaker, is restored, as shown in the final message in the log.

In the provided example, save and restore features are triggered by low-power mode entry and exit events. In the UVM terminology, the events are emitted by the UVM low-power monitor, which observes the low-power interface and uses the Memento design pattern as a client. Saving and restoring multiple configurations is natively supported, therefore the implementation is applicable even in the case such requirement is posed.

```
// Run Mode
m_originator.set_state(run_conf);
`uvm_info("Set Run-Mode configuration", $sformatf("%h", run_conf.fld.get_mirrored_value()), UVM_LOW)
current_conf = m_originator.get_state();
`uvm_info("Current configuration", $sformatf("%h", current_conf.fld.get_mirrored_value()), UVM_LOW)

// LPM entry
m_caretaker.add(m_originator.save_state_to_memento());
`uvm_info("Store Run-Mode configuration", $sformatf("%h", run_conf.fld.get_mirrored_value()), UVM_LOW)
m_originator.set_state(lpm_conf);
`uvm_info("Set LPM configuration", $sformatf("%h", lpm_conf.fld.get_mirrored_value()), UVM_LOW)
current_conf = m_originator.get_state();
`uvm_info("Current configuration", $sformatf("%h", current_conf.fld.get_mirrored_value()), UVM_LOW)

// LPM exit
m_originator.get_state_from_memento(m_caretaker.get(0));
`uvm_info("Restore Run-Mode configuration", $sformatf(""), UVM_LOW);
current_conf = m_originator.get_state();
`uvm_info("Current configuration", $sformatf("%h", current_conf.fld.get_mirrored_value()), UVM_LOW)
```

Figure 2. Memento design pattern – Set, Save and Restore features

```
[Set Run-Mode configuration] 0000000000005a5a
[Current configuration] 0000000000005a5a
[Store Run-Mode configuration] 0000000000005a5a
[Set LPM configuration] 0000000000000000
[Current configuration] 0000000000000000
[Restore Run-Mode configuration]
[Current configuration] 0000000000005a5a
```

Figure 3. Memento design pattern - Output activity log

Figure 4. provides a graphical representation of developed classes and the relations between them, using UML class diagram elements.

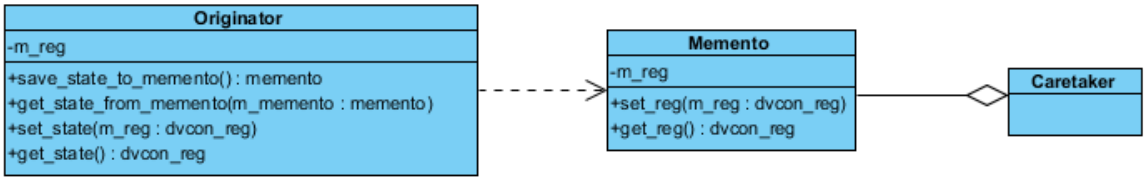

Figure 4. Memento design pattern - UML class diagram

B. Chain of responsibility

Chain of responsibility is a behavioral software design pattern that allows an action or a command to be handled by multiple receivers. Rather than coupling the request sender to all receivers, the sender gives the request to the first element in the chain. The request is successively propagated from one receiver to another, giving more receivers the chance to handle the request. Depending on the application, even if the request is handled by one receiver, it might be passed to the next receiver. In the project example, the design logic reset tree contains three reset levels: soft, medium and hard reset, as displayed in Figure 5. Hard reset represents a power-on reset, controlled by the voltage controller. In case that the input voltage drops under the specified value, power-on reset is asserted. Watchdog controller is configured to track watchdog timer expiration and issue medium level reset. Soft reset is a software reset generated according to the custom register interface towards the functional block. Bus transactions to a certain address within the functional block address space trigger soft reset. Each reset impacts certain registers and fields within the register space. Upon a "harder" reset event, all actions characteristic to a "softer" reset are also applied. The design pattern addresses the register model update, taking into the consideration which register fields are controlled by which reset level.

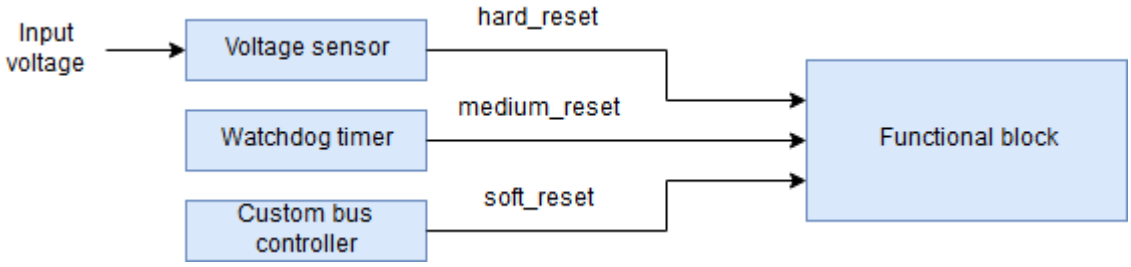

Figure 5. Device reset tree

The components defined by the Chain of responsibility pattern include:

- Abstract handler – Defines features and actions common to each concrete reset handler. It includes the reference to the next handler in the chain. Its API consists of handle() method that handles the request and propagates it to the next handler. Each handler is aware of its severity level and accordingly performs a set of actions in case its severity level is reached. Abstract method reset() is provided as the

interface with which every concrete handler needs to comply. The definition is given in Figure Appendix-4.

The reader should be aware of the fact that the abstract classes are not straightforward to be registered with the UVM factory. The detailed example of their registration is presented in [3]. In the project example, this part is skipped.

- Concrete handlers – For each of the supported reset levels, a dedicated concrete handler is provided, giving implementation to the reset() method and handling register model updates. The mechanism is shown in Figure Appendix-5.
- Client – In the UVM methodology, the dedicated UVM monitor component observes the low-power interface and reset events, using the Chain of responsibility as a client.

The first step when using the infrastructure built according to the Chain of responsibility pattern is configure the order of request handlers. The configuration is shown in Figure 6. It is important to note that the configuration may be dynamically changed and handlers can be added to or removed from the chain.

The main operation is shown in Figure 7, with corresponding output log being presented in Figure 8. It is shown that if medium reset is detected, fields within the register model controlled by soft and medium reset handlers are reset, whereas the field controlled by hard reset handler remains untouched.

The potential drawback of the presented pattern is that it does not assure that a request will be handled by any of the receivers. The propagation of the unhandled requests might be time-consuming and therefore slowing down the simulation.

```
base_handler = m_soft_reset_handler;
m_soft_reset_handler.set_next_handler(m_medium_reset_handler);
m_medium_reset_handler.set_next_handler(m_hard_reset_handler);
```

Figure 6. Chain of responsibility design pattern – Request handlers configuration

```
`uvm_info("", $sformatf("fld soft:   %b", m_reg.fld_soft.get_mirrored_value()),   UVM_LOW)
`uvm_info("", $sformatf("fld medium: %b", m_reg.fld_medium.get_mirrored_value()), UVM_LOW)
`uvm_info("", $sformatf("fld hard:   %b", m_reg.fld_hard.get_mirrored_value()),   UVM_LOW)

base_handler.handle(reset_handler::MEDIUM, m_reg);
`uvm_info("MEDIUM RESET", "", UVM_LOW)
`uvm_info("", $sformatf("fld soft:   %b", m_reg.fld_soft.get_mirrored_value()),   UVM_LOW)
`uvm_info("", $sformatf("fld medium: %b", m_reg.fld_medium.get_mirrored_value()), UVM_LOW)
`uvm_info("", $sformatf("fld hard:   %b", m_reg.fld_hard.get_mirrored_value()),   UVM_LOW)
```

Figure 7. Chain of responsibility design pattern – Medium reset level handling operation

```
[] fld soft:   00000000000000000000000000000000000000000000000000000000000000001
[] fld medium: 00000000000000000000000000000000000000000000000000000000000000001
[] fld hard:   00000000000000000000000000000000000000000000000000000000000000001
[MEDIUM RESET]
[] fld soft:   00000000000000000000000000000000000000000000000000000000000000000
[] fld medium: 00000000000000000000000000000000000000000000000000000000000000000
[] fld hard:   00000000000000000000000000000000000000000000000000000000000000001
```

Figure 8. Chain of responsibility design pattern – Output activity log

The UML diagram of classes defined by the Chain of responsibility design pattern is presented in Figure 9.

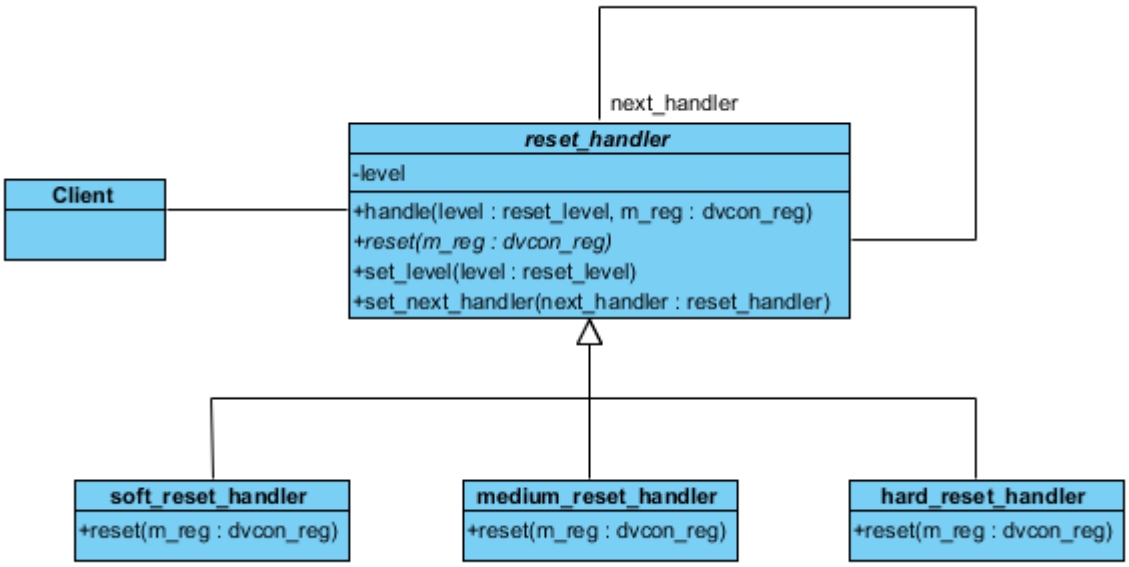

Figure 9. Chain of responsibility – UML class diagram

C. Decorator

Decorator is a structural software design pattern that provides a way of modifying a class object by adding behavior to it, without affecting the other objects of the same class. The pattern allows for adding class functionality without modifying the abstract class. It essentially allows for the base class code to be future-proof for unforeseen modifications. For verification, where features are often added last moment, this feature of the decorator pattern is powerful. The pattern is suitable for modelling complex data items by applying additional set of constraints to them. Its main benefit comparing to class inheritance is that dynamic addition of behavior to objects is achievable. In the project, the technique is used extensively to achieve constrained-random stimulus generation.

To build a Decorator design pattern, it is necessary to define several main parts:

- Base item – Represents the base sequence item with fields to which various constraints shall be applied over the course of a testcase. It is shown in Figure Appendix-6.
- Abstract decorator – Provides a wrapper around the base item, setting up the infrastructure for the concrete decorators. Its code is displayed in Figure Appendix-7.
- Concrete decorators – Each of the concrete decorators provides its own set of constraints, as it can be seen in Figure Appendix-8.

The main idea behind the Decorator design pattern is that various concrete decorators can be active over the course of a simulation, and furthermore, multiple decorators can be active simultaneously. Therefore, at any given moment, any desired combination of constraint sets can be provided to the base item.

The example showing various concrete decorators being wrapped around the base item and affecting the randomization process is in Figure 10. The output log file confirming that adequate constraints have been successfully applied is in Figure 11. The first message shows the outcome of the bare base item randomization, without any applied constraints. It can be seen that all three fields – address, data and delay have a random value within the scope of their legal ranges. Prior to the second randomization, the decorator constraining the address is applied. Therefore, the address is constrained to the range defined by the constraint within the decorator, whereas the data and delay fields still take random values, as seen in the second message. For the third randomization, the data and delay constraining decorators are applied, leaving the address as the only field without applied constraints. Finally, for the last randomization, all three decorators are wrapped around the base item, constraining all three fields.

Despite its advantages, the code in which the Decorator design pattern is overused tends to be hard to maintain, therefore the developer must be careful not to create too many too small classes, increasing the code complexity.

The reader might additionally familiarize themselves with the Decorator design pattern through the UML class diagram shown in Figure 12.

```
base.randomize();
`uvm_info("Randomization outcome",
          $sformatf("addr=%h, data=%h, delay=%d",
                    base.addr, base.data, base.delay), UVM_LOW)

constrain_addr.set_inner(base);
constrain_addr.randomize();
`uvm_info("Randomization outcome",
          $sformatf("addr=%h, data=%h, delay=%d",
                    base.addr, base.data, base.delay), UVM_LOW)

constrain_data.set_inner(base);
constrain_delay.set_inner(constrain_data);
constrain_delay.randomize();
`uvm_info("Randomization outcome",
          $sformatf("addr=%h, data=%h, delay=%d",
                    base.addr, base.data, base.delay), UVM_LOW)

constrain_addr.set_inner(base);
constrain_data.set_inner(constrain_addr);
constrain_delay.set_inner(constrain_data);
constrain_delay.randomize();
`uvm_info("Randomization outcome",
          $sformatf("addr=%h, data=%h, delay=%d",
                    base.addr, base.data, base.delay), UVM_LOW)
```

Figure 10. Decorator design pattern – Randomization

```
[Randomization outcome] addr=ea8453e0, data=52517646, delay=3147082849
[Randomization outcome] addr=00000102, data=7a3ad69c, delay=1720163166
[Randomization outcome] addr=86996f46, data=00000098, delay=        11
[Randomization outcome] addr=00000100, data=00000065, delay=        16
```

Figure 11. Decorator design pattern – Randomization outcome

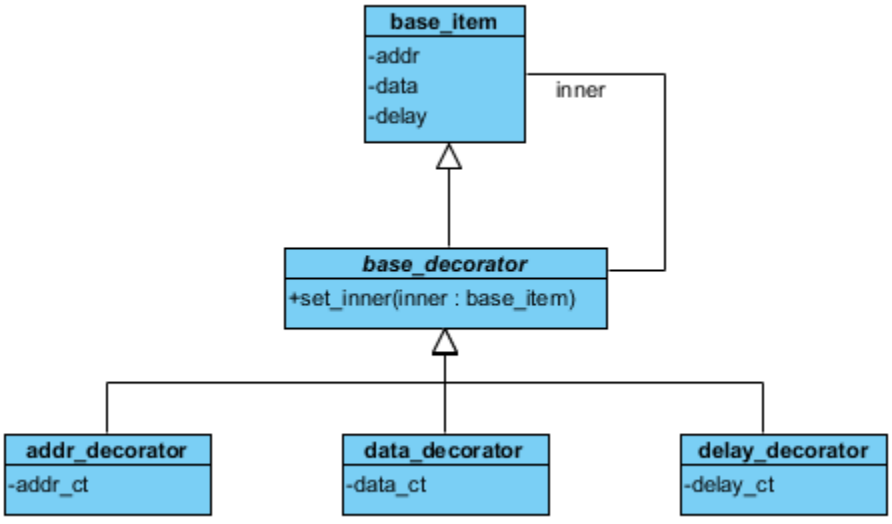

Figure 12. Decorator design pattern – UML class diagram

D. Strategy

The designed logic contains a complex arbitration logic upon the memory access, in which the priority is determined using several algorithms, including fixed-priority and round-robin arbitration. The input to the arbiter are multiple requester signals, as well as the configuration that selects the arbitration type to be applied. According to the selected type and the values of the requester signals, the arbiter determines the source with the highest priority, granting it the right to access the memory, as shown in Figure 13. The arbitration type can be dynamically configured, which is why the feature lends itself to be modelled using Strategy design pattern, in which a certain algorithm to be applied can be selected at run-time out of a family of provided algorithms. Additionally, in case it proves required, it is straightforward to add new algorithms for the arbitration. It is worth noting the Decorator design pattern is also used to dynamically change behavior. The key difference is that, while the Decorator is used to add to the existing core functionality, the Strategy targets the change in the core functionality itself. To quote, the StackOverflow: "Strategy lets you change the guts of an object. Decorator lets you change the skin." [4]

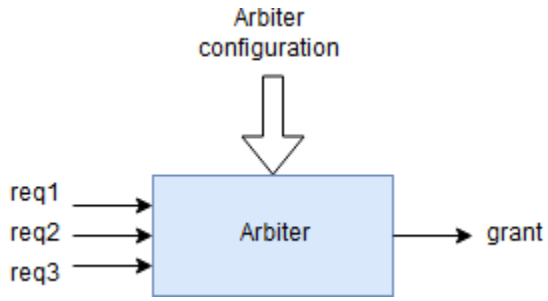

Figure 13. Arbitration logic block diagram

The model compliant with Strategy design pattern guidelines includes the following elements:

- Strategy – Defines a general set of operations that the model should support. In the project, the Strategy defines a general operation of arbitration, which can be performed in several different ways. The abstract Strategy class is shown in Figure Appendix-9.
- Concrete Strategy – For each of the necessary algorithms, a Concrete Strategy class is defined, providing the implementation to the certain algorithm. Wrapping each algorithm into a separate class improves code readability and extensibility. Various arbitration rules implemented in corresponding classes are displayed in Figure Appendix-10.
- Context – Context is a class that uses the Strategy. In the example, the UVM Scoreboard component, relying upon the Strategy, checks that the arbitration logic within the device is implemented correctly. The code extract in Figure Appendix-11 shows the Context class.

A simple usage example is shown in Figure 14. with the corresponding output log being in Figure 15. In the example, the arbitration winner is the one with the highest index, as indicated by configuring *strategy_high_prio* arbitration type. Since the active sources are the ones with index 2 and 3, the expected arbitration winner is the one with index 3, as determined by executing the strategy. The predicted arbitration winner can be compared against the grant signal within the design, to determine whether the designed arbitration logic is implemented correctly. Supposing the configuration changed, it would simply be required to update the context with a new strategy of a different corresponding type.

The UML class diagram is shown in Figure 16.

```
m_strategy  = strategy_high_prio::type_id::create("m_strategy");
m_context.set_strategy(m_strategy);

req_q[REQ1] = 0;
req_q[REQ2] = 1;
req_q[REQ3] = 1;
arb_winner = m_context.execute_strategy(req_q);
`uvm_info("Arbitration winner: ", $sformatf("%s", arb_winner.name()), UVM_LOW);
```

Figure 14. Strategy design pattern – Usage example: Arbitration logic modelling

```
@ 0: reporter [Arbitration winner: ] REQ3
```

Figure 15. Strategy design pattern – Usage example: Arbitration logic modelling output

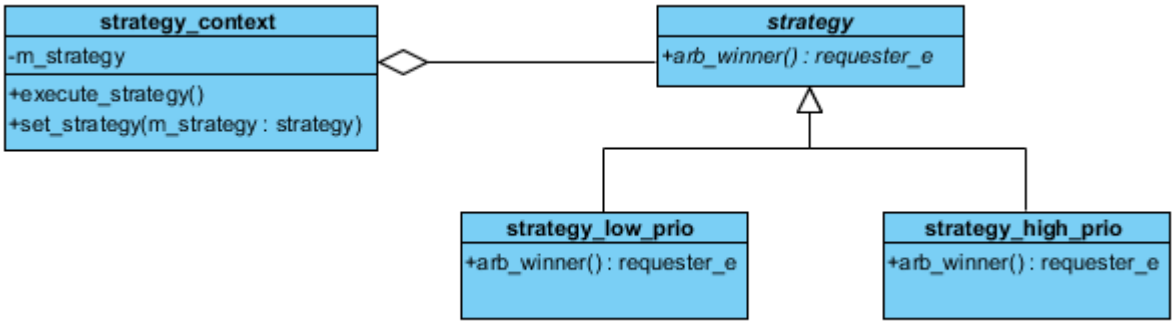

Figure 16. Strategy design pattern – UML class diagram

E. Singleton

Singleton is a simple creational design pattern restricting the number of class objects that can be instantiated and providing the global access to the objects. The environment configuration class and the timeout logic handling class are implemented as singletons. Each component in the environment that needs to monitor for the timeout event on a certain interface during a certain scenario, can rely on the timeout handling class to determine the moment of the timeout expiration. Assuring that all components access the same object of the timeout handling class facilitates debugging.

The design pattern defines two main components:

- Singleton class – The class of which only a limited number of objects, typically one, can be created. To achieve this, the constructor method of the class is defined as protected. The object is accessed using the static Instance() method, which performs "lazy initialization" upon its first call. The example is shown in Figure Appendix-12.
- Client – Any UVM component may use the Singleton's Instance() method to get access to the class object.

The main drawback is that a single class performs two separate tasks, taking care of its initialization as well as its core functionality.

The example code showing the Singleton design pattern usage is shown in Figure 17, with the corresponding output presented in Figure 18. The output log shows that the timeout handling class is capable of tracking multiple timeout events simultaneously, signalizing the expiration of the first timeout – at time = 500, and later of the second timeout – at time = 1000. The UML class diagram is given in Figure 19.

```
timeout inst = timeout::Instance();
fork
  begin
    `uvm_info("START TIMEOUT COUNT 1", "", UVM_LOW)
    inst.wait_timeout(500);
    `uvm_info("END TIMEOUT COUNT 1", "", UVM_LOW)
  end
  begin
    `uvm_info("START TIMEOUT COUNT 2", "", UVM_LOW)
    inst.wait_timeout(1000);
    `uvm_info("END TIMEOUT COUNT 2", "", UVM_LOW)
  end
join
```

Figure 17. Singleton design pattern – Usage example

```
UVM_INFO testbench.sv(32) @ 0: reporter [START TIMEOUT COUNT 1]
UVM_INFO testbench.sv(37) @ 0: reporter [START TIMEOUT COUNT 2]
UVM_INFO testbench.sv(34) @ 500: reporter [END TIMEOUT COUNT 1]
UVM_INFO testbench.sv(39) @ 1000: reporter [END TIMEOUT COUNT 2]
```

Figure 18. Singleton design pattern – Usage example output log

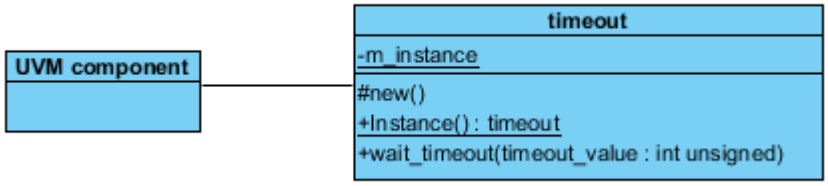

Figure 19. Singleton design pattern – UML class diagram

F. Template method

Template method defines a set of operations to be performed orderly, leaving the implementation of some steps to the derived classes while maintaining the overall algorithm structure. The technique is commonly exploited to provide pre-processing and post-processing hooks to a main operation. While it is already used massively in the UVM library (e.g. *pre_body* and *post_body* hooks to a *body* method of *uvm_sequence*), in a similar way it is employed in monitor components, to allow the main monitor operation to be further extended in the project:

- Base class – Base monitor component defines non-virtual *collect_transactions()* template method, providing empty pre-processing and post-processing hooks. The component is shown in Figure Appendix-13.
- Extended class – Provides project-specific operation, by implementing the provided hooks, as seen in Figure Appendix-14.

To replace an object of the base class with an object of an extended class, the developer utilizes UVM Factory override concept. Output log files without and with applied override, demonstrating the Template method pattern are shown in Figure 20. and 21, respectively. In the former, the messages are produced by the base monitor with empty *pre_collect()* and *post_collect()* hooks. In the latter, the instance of the extended monitor is used instead, with implemented hooks that can be used for project-specific checking and similar project-specific features, without modifying the base monitor code. The main *collect()* operation is inherited from the base monitor and, therefore, remains the same in both cases, producing the same output. The template method *collect_transactions()* assures that the hooks will be invoked at the appropriate spots, before and after the main operation, both in the base and the extended component.

A very common approach is defining the base class as abstract, with abstract methods that the extended classes should implement.

The main drawback of the described pattern is that it may complicate the debugging process in case that both base and extended class implement complex logic.

The simple UML diagram is displayed in Figure 22.

```
@ 0: uvm_test_top.m_env.m_agent.m_mon [PRE_COLLECT] Empty method
@ 0: uvm_test_top.m_env.m_agent.m_mon [COLLECT] Collect item start
@ 100000: uvm_test_top.m_env.m_agent.m_mon [COLLECT] Collect item end
@ 100000: uvm_test_top.m_env.m_agent.m_mon [POST_COLLECT] Empty method
```

Figure 20. Template method design pattern – Base monitor with empty hooks: usage output log

```
0: uvm_test_top.m_env.m_agent.m_mon [PRE_COLLECT] Pre: Collect item
0: uvm_test_top.m_env.m_agent.m_mon [COLLECT] Collect item start
100000: uvm_test_top.m_env.m_agent.m_mon [COLLECT] Collect item end
100000: uvm_test_top.m_env.m_agent.m_mon [POST_COLLECT] Post: Collect item
```

Figure 21. Template method design pattern – Extended monitor with implemented hooks: usage output log

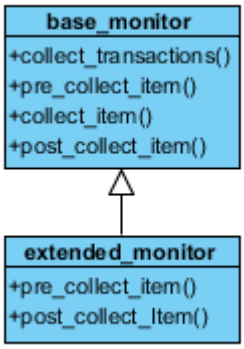

Figure 22. Template method design pattern – UML class diagram

G. Iterator

The verification environment includes a complex scoreboarding mechanism, which operation underlies upon modifications performed on the transactions collected by the interface monitors. For example, a dedicated configuration register field is used to enable or disable ECC calculation. Upon its change, it is necessary to iterate through the transactions within the scoreboard and modify the values of collected data, to achieve correct prediction. Any scenario in which traversing is performed upon a collection of objects, regardless of its internal structure, is suitable to be modelled using the Iterator design pattern. The main advantage of the solution is that the internal structure storing the data is not exposed and can therefore be modified without affecting the rest of the environment. The usage of Iterator design pattern typically increases the flexibility, without any major drawback.

The design pattern defines several key components:

- Abstract Iterator – General Iterator that defines the operations necessary to iterate through a collection. It is shown in Figure Appendix-15.
- Abstract Container – Defines a general collection that can be iterated with an Iterator. An example is presented in Figure Appendix-16.
- Concrete Iterator – Implements the "iterate" operation through a given data structure, such as queue, array, linked list, and more general, any structure that can contain data. It is displayed in Figure Appendix-17.
- Concrete Container – Represents a wrapper around a certain data structure and creates a Concrete Iterator that can iterate the data structure, as shown in Figure Appendix-17.

The usage example that shows how to add items to the data container in the scoreboard, and later iterate the container, performing actions upon the collected items, is shown in Figure 23, whereas the corresponding output is displayed in Figure 24. The output log shows that for each item within the data container, the method *recalculate_data()* is invoked, updating the data used for prediction and producing the output message.

The UML class diagram showing the relations between the presented classes is in Figure 25.

```
data_cont.add(item1);
data_cont.add(item2);
data_cont.add(item3);
data_cont.add(item4);
data_cont.add(item5);

...

for(iterator iter = data_cont.get_iterator(); iter.has_next();) begin
  item = iter.next();
  item.recalculate_data(ECC_ON);
end
```

Figure 23. Iterator design pattern – Iteration through a collection of items

```
reporter@@item1 [item1] Data recalculation due to ECC bit change- ECC bit is: ECC_ON
reporter@@item2 [item2] Data recalculation due to ECC bit change- ECC bit is: ECC_ON
reporter@@item3 [item3] Data recalculation due to ECC bit change- ECC bit is: ECC_ON
reporter@@item4 [item4] Data recalculation due to ECC bit change- ECC bit is: ECC_ON
reporter@@item5 [item5] Data recalculation due to ECC bit change- ECC bit is: ECC_ON
```

Figure 24. Iterator design pattern – Iteration outcome

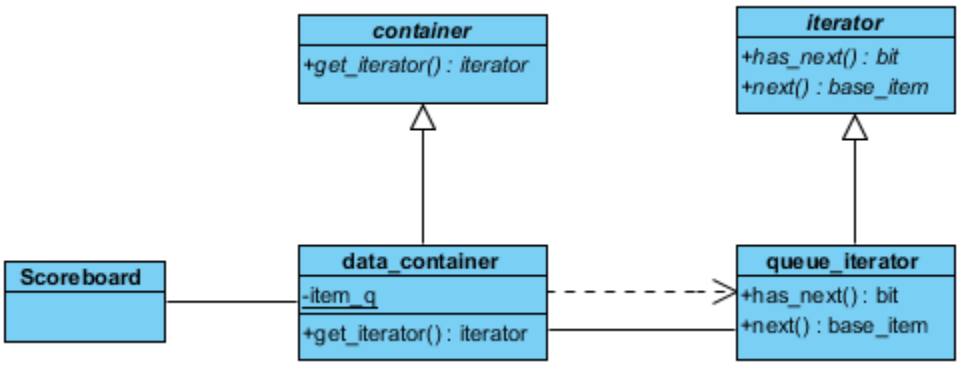

Figure 25. Iterator design pattern – UML class diagram

H. Other utilized patterns

Apart from all aforementioned design patterns, which were discussed in detail, several other design patterns are utilized within the verification environment in the project. Here, they will be briefly introduced.

Factory is a design pattern already implemented in the UVM library, to create an object of the requested type. [5] The pattern is massively used in the project. For example, by wrapping covergroups within UVM wrapper object, the UVM Factory is used to achieve coverage creation on demand, dynamically controlling the coverage model to be instantiated within a certain testcase. The implementation is in detailed discussed in [6].

Next design pattern that is utilized is Observer. While the Observer design pattern already finds its application within a common UVM-compliant verification environment, through the usage of TLM analysis ports, it enhances the class communication even further in the project. It is utilized to achieve dynamic connections between the components, as opposed to static connections, offered by TLM analysis ports. The solution heavily relies upon the implementation presented in [7]. The solution targets another major limitation of TLM analysis ports, as the interface between the class publisher and the class listener is not limited to a single transaction type.

The combination of State and Mediator design pattern proved suitable for modelling the finite-state machines within the designed logic. The designed logic consists of several finite-state machines, which extensive verification is a part of the functional verification process. To facilitate the checking of the logic correctness and to incorporate FSM related parameters into the functional coverage metric, it is suitable to develop a state machine reference model. The methodology describing how to model the static component of the state machine (states) using State

pattern, and the dynamic component of the state machine (state transitions) using Mediator pattern is in detail presented in [8]. The proposed implementation improves code readability and reusability on both active generation side and passive checking and coverage collection side. It also facilitates the maintainability, by localizing the changes upon the addition of new FSM states.

Visitor is a design pattern used to add a new operation to each class in an existing class structure, suitable to be implemented for classes within a well-defined class hierarchy, such is the UVM based environment. It is utilized to perform reporting over the course of a simulation. The infrastructure for the usage of Visitor is provided within UVM library and is described in [9].

IV. Summary

Design patterns are a proven solution, targeting code reusability and extensibility, as well as facilitating its maintenance. A code based on design patterns, accompanied with UML class diagrams, is more readable and easier to document, as the role of certain components and the relation between them is straightforward to understand. Many challenges encountered within a verification environment are suitable to be overcome using a certain design pattern, as shown in the given examples, resulting in the improved code quality and reduced development effort.

The potential room for code improvement is to incorporate SystemVerilog interface classes [7], which are a common construct in the software development world, but are yet to be fully utilized in the verification. The authors' main goal was to come up with fully UVM-compliant solution, and they, therefore, mainly relied upon the language elements that are straightforward to integrate into the codebase developed according to the UVM methodology.

References

[1] E. Gamma, R. Helm, R. Johnson and J. Vlissides, "Design Patterns: Elements of Reusable Object-Oriented Software" Addison-Wesley, 1994.
[2] http://www.uml.org/
[3] http://blog.verificationgentleman.com/2016/02/abstract-classes-uvm-factory.html, Tudor Timisescu, referenced on 1. 11. 2017.
[4] https://stackoverflow.com/questions/26422884/strategy-pattern-v-s-decorator-pattern
[5] Accellera, "UVM User Guide, v1.1", page 131, www.uvmworld.org
[6] D. Tomušilović, "Functional Coverage of Register Access via Serial Bus Interface using UVM", DVCon U.S. 2017.
[7] S. Sokorac, "SystemVerilog Interface Classes – More Useful Than you Thought", DVCon U.S. 2016.
[8] D. Tomušilović, M. Minović, "Modelling Finite-State Machines in the Verification Environment using Software Design Patterns", DVCon Europe 2017.
[9] https://verificationacademy.com/verification-methodology-reference/uvm/docs_1.2/html/files/base/uvm_traversal-svh.html

APPENDIX

A. *Memento*

```
class memento extends uvm_object;

  `uvm_object_utils(memento)

  local dvcon_reg m_reg;

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

  function void set_reg(dvcon_reg m_reg);
    this.m_reg = m_reg;
  endfunction

  function dvcon_reg get_reg();
    return m_reg;
  endfunction
endclass
```

Figure Appendix-1. Memento design pattern - Memento class

```
class originator extends uvm_object;

  `uvm_object_utils(originator)

  local dvcon_reg m_reg;

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

  function void set_state(dvcon_reg m_reg);
    this.m_reg = m_reg;
  endfunction

  function dvcon_reg get_state();
    return m_reg;
  endfunction

  function memento save_state_to_memento();
    memento m_memento;
    m_memento = memento::type_id::create("m_memento");
    m_memento.set_reg(m_reg);
    return m_memento;
  endfunction

  function void get_state_from_memento(memento m_memento);
    m_reg = m_memento.get_reg();
  endfunction
endclass
```

Figure Appendix-2. Memento design pattern - Originator class

```
class caretaker extends uvm_object;

  `uvm_object_utils(caretaker)

  local memento memento_q[$];

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

  function void add(memento state);
    memento_q.push_back(state);
  endfunction

  function memento get(int index);
    return memento_q[index];
  endfunction
endclass
```

Figure Appendix-3. Memento design pattern - Caretaker class

B. Chain of responsibility

```
virtual class reset_handler extends uvm_object;

  // Abstract classer are not straightforward to be registered
  // with UVM factory
  // `uvm_object_utils(reset_handler) cannot be utilized

  typedef enum { HARD = 0, MEDIUM = 1, SOFT = 2} reset_level;

  protected int level;

  protected reset_handler next_handler;

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

  function void set_level(reset_level level);
    this.level = level;
  endfunction

  function void set_next_handler(reset_handler next_handler);
    this.next_handler = next_handler;
  endfunction

  function void handle(reset_level level, dvcon_reg m_reg);
    if(this.level >= level) begin
      reset(m_reg);
    end
    if ((this.level > level) && (next_handler != null)) begin
        next_handler.handle(level, m_reg);
    end
  endfunction

 pure virtual protected function void reset(dvcon_reg m_reg);
endclass
```

Figure Appendix-4. Chain of responsibility design pattern – Abstract handler

```
class soft_reset_handler extends reset_handler;

  `uvm_object_utils(soft_reset_handler)

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

  protected function void reset(dvcon_reg m_reg);
    m_reg.fld_soft.reset();
  endfunction
endclass

class medium_reset_handler extends reset_handler;

  `uvm_object_utils(medium_reset_handler)

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

  protected function void reset(dvcon_reg m_reg);
    m_reg.fld_medium.reset();
  endfunction
endclass

class hard_reset_handler extends reset_handler;

  `uvm_object_utils(hard_reset_handler)

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

  protected function void reset(dvcon_reg m_reg);
    m_reg.fld_hard.reset();
  endfunction
```

Figure Appendix-5. Chain of responsibility design pattern – Concrete handlers

C. Decorator

```
  class base_item extends uvm_sequence_item;

    rand bit[31:0]     addr;
    rand bit[31:0]     data;
    rand int unsigned delay;

    `uvm_object_utils_begin(base_item)
      `uvm_field_int(addr, UVM_DEFAULT)
      `uvm_field_int(data, UVM_DEFAULT)
      `uvm_field_int(delay, UVM_DEFAULT)
    `uvm_object_utils_end

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

Figure Appendix-6. Decorator design pattern – Base item

```
virtual class base_decorator extends base_item;
  rand base_item inner;

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

  function void set_inner(base_item inner);
    this.inner = inner;
  endfunction

  constraint inner_ct { inner.addr == addr; inner.data == data; inner.delay == delay; }
endclass
```

Figure Appendix-7. Decorator design pattern – Abstract base decorator

```
class delay_decorator extends base_decorator;

  `uvm_object_utils(delay_decorator)

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

  constraint delay_ct { delay inside {[10:20]}; }
endclass

class addr_decorator extends base_decorator;

  `uvm_object_utils(addr_decorator)

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

  constraint addr_ct { addr inside {['h100:'h108]}; }
endclass

class data_decorator extends base_decorator;

  `uvm_object_utils(data_decorator)

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

  constraint data_ct { data inside {['h5a:'ha5]}; }
endclass
```

Figure Appendix-8. Decorator design pattern – Concrete decorators

D. Strategy

```
virtual class strategy extends uvm_object;

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

  pure virtual function requester_e arb_winner(ref bit req_assoc[NUM_OF_REQ]);
endclass
```

Figure Appendix-9. Strategy design pattern – Abstract Strategy class

```
class strategy_low_prio extends strategy;

  `uvm_object_utils(strategy_low_prio)

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

  virtual function requester_e arb_winner(ref bit req_assoc[NUM_OF_REQ]);
    for (int i = 0; i < NUM_OF_REQ; i++) begin
      if (req_assoc[i] == 1) begin
        return requester_e'(i);
      end
    end
    return NONE;
  endfunction
endclass

class strategy_high_prio extends strategy;

  `uvm_object_utils(strategy_high_prio)

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

  virtual function requester_e arb_winner(ref bit req_assoc[NUM_OF_REQ]);
    for (int i = 0; i < NUM_OF_REQ; i++) begin
      if (req_assoc[NUM_OF_REQ-1-i] == 1) begin
        return requester_e'(NUM_OF_REQ-1-i);
      end
    end
    return NONE;
  endfunction
endclass
```

Figure Appendix-10. Strategy design pattern – Concrete Strategy classes implementing various arbitration mechanisms

```
class strategy_context extends uvm_object;

  `uvm_object_utils(strategy_context)

  local strategy m_strategy;

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

  function void set_strategy(strategy m_strategy);
    this.m_strategy = m_strategy;
  endfunction

  function requester_e execute_strategy(ref bit req_q[NUM_OF_REQ]);
    return m_strategy.arb_winner(req_q);
  endfunction
endclass
```

Figure Appendix-11. Strategy design pattern – Context class executing a Strategy

E. Singleton

```
class timeout;

  local static timeout m_instance = null;

  protected function new();
  endfunction

  static function timeout Instance();
    if (m_instance == null)
      m_instance = new();
    return m_instance;
  endfunction

  task wait_timeout(int unsigned timeout_value);
    #(timeout_value*1ns);
  endtask
endclass
```

Figure Appendix-12. Singleton design pattern – Timeout logic handler implemented as Singleton

F. Template method

```
class base_monitor extends uvm_monitor;

  `uvm_component_utils(base_monitor)

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

  virtual task pre_collect_item();
    `uvm_info("PRE_COLLECT", "Empty method", UVM_LOW)
  endtask

  virtual task post_collect_item();
    `uvm_info("POST_COLLECT", "Empty method", UVM_LOW)
  endtask

  virtual task collect_item();
    `uvm_info("COLLECT", "Collect item start", UVM_LOW)
    ...
    `uvm_info("COLLECT", "Collect item end", UVM_LOW)
  endtask

  task collect_transactions();
    pre_collect_item();
    collect_item();
    post_collect_item();
  endtask

  ...
endclass
```

Figure Appendix-13. Template method design pattern – Base monitor with empty hooks

```
class extended_monitor extends base_monitor;

  `uvm_component_utils(extended_monitor)

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

  virtual task pre_collect_item();
    `uvm_info("PRE_COLLECT", "Pre: Collect item", UVM_LOW)
  endtask

  virtual task post_collect_item();
    `uvm_info("POST_COLLECT", "Post: Collect item", UVM_LOW)
  endtask
endclass
```

Figure Appendix-14. Template method design pattern – Extended monitor with implemented hooks

G. Iterator

```
virtual class iterator extends uvm_object;

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

  pure virtual function bit   has_next();
  pure virtual function base_item next();
endclass
```

Figure Appendix-15. Iterator design pattern – Abstract Iterator

```
virtual class container extends uvm_object;

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

  pure virtual function iterator get_iterator();
endclass
```

Figure Appendix-16. Iterator design pattern – Abstract Container with Iterator

```
class data_container extends container;
  `uvm_object_utils(data_container)

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

  static base_item item_q[$];

  class queue_iterator extends iterator;

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

    int index;

    virtual function bit has_next();
      if(index < item_q.size()) begin
          return 1;
      end
      return 0;
    endfunction

    virtual function base_item next();
      if(this.has_next()) begin
          return item_q[index++];
      end
      return null;
    endfunction
  endclass

  virtual function iterator get_iterator();
    queue_iterator queue_it = queue_iterator::type_id::create("queue_it");
    return queue_it;
  endfunction
  ...
endclass
```

Figure Appendix-17. Iterator design pattern – Data container with Queue iterator