Design Patterns by Example for SystemVerilog Verification Environments Enabled by SystemVerilog 1800-2012

Eldon Nelson M.S. P.E.
Intel Corporation (eldon_nelson@ieee.org)

***Abstract*- "Design Patterns", published in 1994, is widely seen as popularizing the idea of software design patterns. The book contained explanations and applications of software design patterns and gave them their definitive names. The missing component to actual implementation of many of the design patterns in SystemVerilog is language support, which has only recently become available with the release of the SystemVerilog 1800-2012 specification. This paper will realize design pattern examples from the book "Head First Design Patterns"—originally written in Java—and port them to SystemVerilog while being as true to the original implementation as possible. The goal will be to expose that many design patterns are now possible to implement in SystemVerilog 1800-2012. Applications of those design patterns tailored to verification environments is demonstrated to show its usefulness.**

I. Introduction

Modern verification environments created in a framework such as UVM encapsulate lessons learned from years of software design experimentation by the computer science community. However, there are cornerstones of software design that are not widely deployed in UVM verification environments. These missing software design lessons make it harder to create flexible verification environments capable of meeting current verification needs.

The book "Design Patterns" [1] published in 1994 is widely seen as popularizing the idea of software design patterns. The book contained explanations and applications of software design patterns and gave them their definitive names. UVM borrows from "Design Patterns" for some aspects of its design. For example, the UVM Factory conforms to many of the ideas of the Factory Pattern described in "Design Patterns".

The missing component to actual implementation of many of the design patterns is language support, which has only recently become available with the release of the SystemVerilog 1800-2012 specification [2]. Without language support, many of the design patterns are difficult to recognize or are impossible to implement properly. For example, the SystemVerilog construct "implements" (Figure 1) is crucial for executing the Strategy Pattern and is critical for executing the software design construct known as "composition". A recurring theme in many of the design patterns is a robust use of composition, which was not straightforward to implement in previous SystemVerilog releases. Without composition, there is a tendency to create an unnecessary and gratuitous class hierarchy, which is more fragile to maintain and less extensible.

```
class MuteQuack implements QuackBehavior;
   virtual function void quack();
      $display("<< Silence >>");
   endfunction
endclass
```

FIGURE 1. EXAMPLE OF SYSTEMVERILOG "IMPLEMENTS"

The code in Figure 1 is SystemVerilog, but is valid only with the 1800-2012 version of the specification because the keyword "implements" is only defined there. A user of SystemVerilog would be familiar with the keyword "extends" in the place of "implements". However, the two keywords have very different meanings despite being in the same keyword position. The keyword "extends" means to use inheritance to extend from a base class. Inheritance has many useful applications in object-oriented programming, but in many SystemVerilog designs inheritance is seen

as the only method that can change or add new functionality to a class. The keyword "implements" is a piece of an alternative method of changing the functionality of a class called composition.

This paper will realize design pattern examples from the book "Head First Design Patterns" [3]—originally written in Java—and port them to SystemVerilog while being as true to the original as possible. The goal will be to expose that it is now possible to implement many design patterns in SystemVerilog 1800-2012.

Five design pattern examples: the Strategy Pattern [4] [5], the Observer Pattern [6] [7], the Decorator Pattern [8], the Singleton Pattern [9] and the State Pattern [10] are implemented in SystemVerilog and are tested with three leading SystemVerilog simulators; the examples have an exceedingly high fidelity with their original Java counterparts. The developed code examples [11] are released with a GNU GPLv2 [12] license. There are some deficiencies in specific simulation vendor tool support while the vendors work to fully implement the required new SystemVerilog 1800-2012 language constructs. These implementation hurdles are covered and categorized. New capabilities in the SystemVerilog specification are beginning to open to the verification community a world to write expressive verification environments that can now borrow more ideas codified in "Design Patterns."

II. Approach

Using examples from a well-regarded book on the topic of Design Patterns, with consent from the publishers and authors, helps to give context and available resources for learning more. The code examples used in this paper are taken from the book "Head First Design Patterns", which uses the programming language Java in its examples. There is also a video course "Foundations of Programming: Design Patterns" [13] on the self-learning website Lynda.com [14] presented by the authors of "Head First Design Patterns" that covers the first part of that book. The author of this paper, Eldon Nelson, has no personal or financial connections with the authors, book publisher or Lynda.com.

This paper gives key code excerpts for certain concepts, but for full understanding, the complete working code examples are available at the URL below, and specifically through the Reference numbers provided.

```
https://github.com/tenthousandfailures/systemverilog-design-patterns
```

The code examples provided have been tested on the Cadence [15], Mentor [16] and Synopsys [17] simulators. To make it easy to follow along and experiment, in each example directory there is a run script for each simulator along with the expected simulation text output of the run. The run script also makes a note on the simulator compatibility of the particular example.

III. Composition with SystemVerilog

The Strategy Pattern is the first design pattern described in "Head First Design Patterns". The Strategy Pattern is one of the most basic applications of an object-oriented construct called composition. It might be helpful to quote a paragraph from "Design Patterns" explaining the difference between composition versus the more familiar inheritance used in SystemVerilog verification environments today. "Object composition is an alternative to class inheritance. Here, new functionality is obtained by assembling or composing objects to get more complex functionality. Object composition requires that the objects being composed have well-defined interfaces. This style of reuse is called black-box reuse, because no internal details of objects are visible. Objects appear only as 'black boxes'" [1, p. 19]. Forms of composition would have been possible in releases of SystemVerilog before 1800-2012, but the implementation would be missing the use of the "interface class" and "implements" keywords that would clearly and consistently realize the idea of composition.

```
public interface QuackBehavior {
   public void quack();
}
```

Figure 2. Java version of interface class from QuackBehavior.java [18]

```
interface class QuackBehavior;
   pure virtual function void quack();
endclass
```

Figure 3. SystemVerilog Version of interface class from QuackBehavior.sv [4]

We can compare the code examples above from “Head First Design Patterns”, showing how to translate the idea of creating an interface class from Java to SystemVerilog. The description from “Design Patterns” mentions having a “well-defined interface[s]” [1, p. 19]. The SystemVerilog “interface class” in Figure 3 defines a class interface named “QuackBehavior” that implements a function called “quack”. The interface class is a contract that defines the functions (and more), that the interface class must provide when implemented. Attributes of the interface class function such as its return type and its exact arguments are defined. The SystemVerilog 1800-2012 specification is explicit in stating that “an interface class shall only contain pure virtual methods” [2, p. 157] for functions, which is why it is compulsory to add the “pure virtual” attribute to our function declaration.

To implement a class that actually performs the real function of our defined class interface, we use the keyword “implements”. The code of the Java implementation in Figure 4 and of the SystemVerilog implementation in Figure 5 is similar, and can be easily translated between the two languages.

```
public class MuteQuack implements QuackBehavior {
   public void quack() {
      System.out.println("<< Silence >>");
   }
}
```

FIGURE 4. MUTEQUACK.JAVA [18]

```
class MuteQuack implements QuackBehavior;
   virtual function void quack();
      $display("<< Silence >>");
   endfunction
endclass
```

FIGURE 5. MUTEQUACK.SV [4]

With the interface class (the contract) and the implementation of the interface class, we have the basis for doing a formal implementation of composition. We can define functions with all of their attributes in an abstract way with an interface class. Then, we can implement multiple variations of those implementations and be certain that we can swap these interchangeable implementations at will. There is a wealth of opinions on the topic of composition versus inheritance [19] and there is even an idiom “favor object composition over class inheritance” [1, p. 20] as the second principle of object-oriented design from “Design Patterns”.

IV. Strategy Pattern with the Duck Simulator

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

- Definition of the Strategy Pattern from "Design Patterns" [1, p. 135]

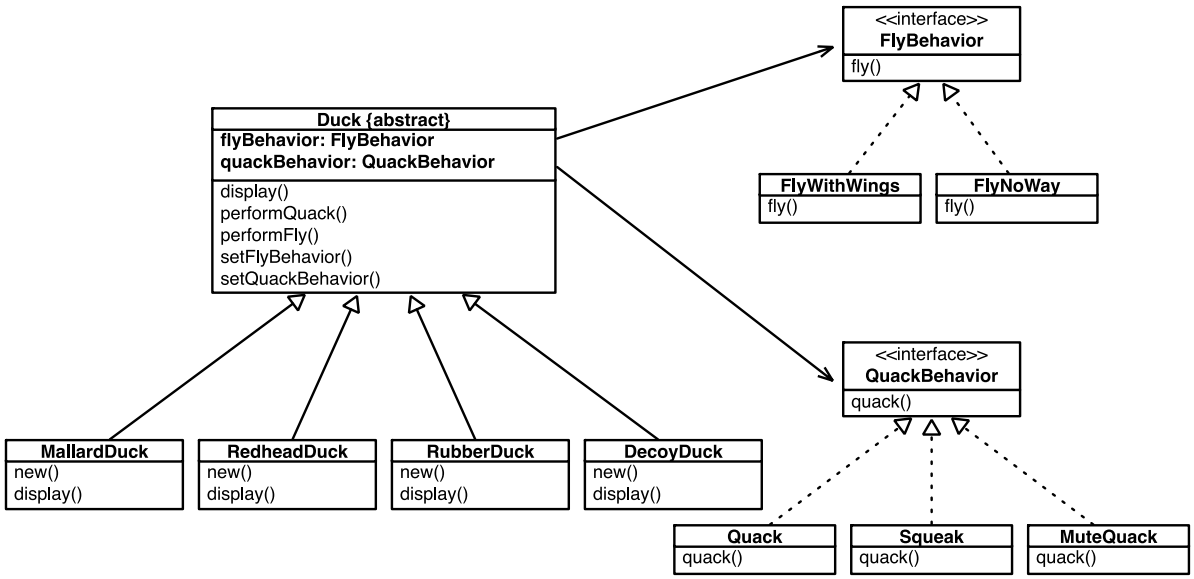

Figure 6. Strategy Pattern UML Class Diagram for Duck Simulator

The example described in "Head First Design Patterns" for the Strategy Pattern describes a Duck Simulator. The Duck object can perform two actions: fly and quack. There are many different types of ducks, such as the Mallard Duck, which can fly with wings and quack as a duck does. And, there are ducks such as the Rubber Duck, which does not fly and squeaks instead of quacks. The UML (Unified Modeling Language [20]) class diagram in Figure 6 shows the types of ducks to be simulated and their possible quack and fly behaviors. The interface classes denoted with "<<interface>>" in the UML class diagram in Figure 6 implements, through indirection combined with composition, the quack and fly behaviors.

```
class RubberDuck extends Duck;

   function new();
      FlyNoWay f = new();
      Squeak q = new();
      setFlyBehavior(f);
      setQuackBehavior(q);
   endfunction

   virtual function void display();
      $display("I'm a rubber duckie");
   endfunction

endclass
```

Figure 7. Strategy Pattern Example File RubberDuck.sv [4]

The finished code example [4] and the excerpt from Figure 7 shows how composition allows for dynamically swapping out and mixing the behavioral functions at runtime. In Figure 7 we see that the RubberDuck class is created by

mixing in the “FlyNoWay” behavior and a “Squeak” behavior. This is done in the constructor of RubberDuck, but there is no problem in changing this behavior during runtime. A benefit to this method is that the two behaviors FlyBehavior and QuackBehavior can be reused in other unrelated classes or mixed in any way needed as new types of Ducks are required. It also means that new behaviors can easily be created without unintended consequences because there is a clear class interface that describes how these functions need to work. If this example were done without composition or without the Strategy Pattern, a complicated, and ultimately contradictory, inheritance hierarchy would have to be created to resolve the inheritance of the two behaviors and future new behaviors.

A constraint of using the Strategy Pattern and interfaces classes is that the behaviors do not have access to the Duck class variables. The behaviors only have passed to them exactly what is needed for them to accomplish their purpose. Like the Java version of “interface”, the SystemVerilog “class interface” (the name “class interface” was used in the SystemVerilog implementation because “interface” was already used for a different construct in the language) does not make available the variables of the object to which it is connected. Having the Behaviors be self-contained reduces the dependencies they have and makes them then easier to maintain, because they have a defined interface and cannot reach into other objects unless they are passed those handles explicitly.

STRATEGY PATTERN APPLIED TO VERIFICATION

A common problem in verification is the changing encapsulation format; an example would be the idea of a packet class. A packet might have different versions, such a v1, v2, v3, that evolve over time and hardware cycles that are largely the same but vary in their packing, or perhaps even the number of bits in one of its fields. The packet class versioning problem is typically handled by inheritance, but there are properties of using the Strategy Pattern that are attractive for this type of problem.

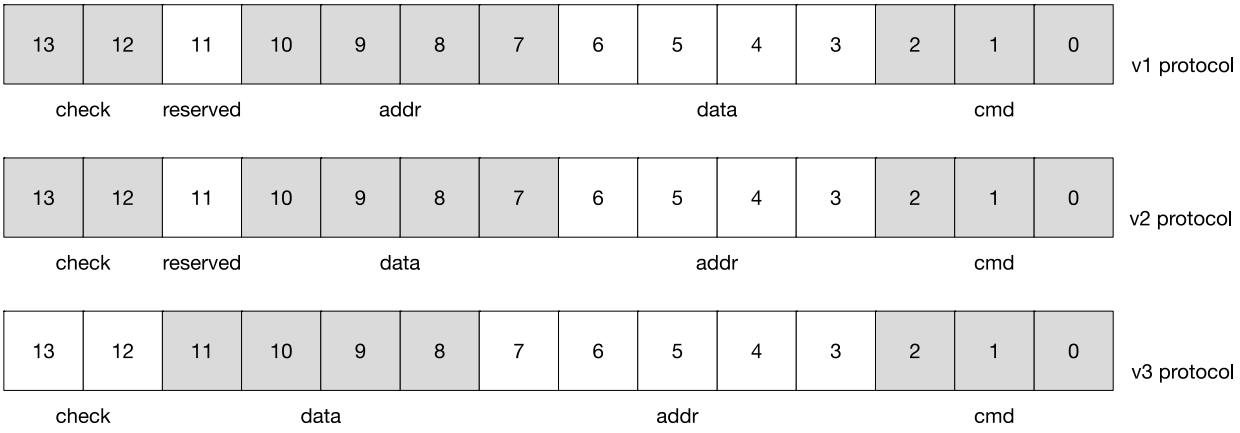

FIGURE 8. PACKET PROTOCOL LAYOUT

Using the packet protocols in Figure 8, from v1 to v3 we see that the locations of “addr” and “data” change, and that the number of bits for “addr” increases in the v3 protocol. Also, the “check” bits are user-selectable between a simple parity check or a CRC within the same protocol. If we wanted to apply the Strategy Pattern to this problem, the UML class diagram in Figure 9 could be one possible implementation. It uses a separate class called Fields to abstract the field contents from its bit level representation. The packets rely on two Behaviors, PackBehavior and CheckBehavior, and their “pack” and “unpack” functions to convert to and from the bit-level representations.

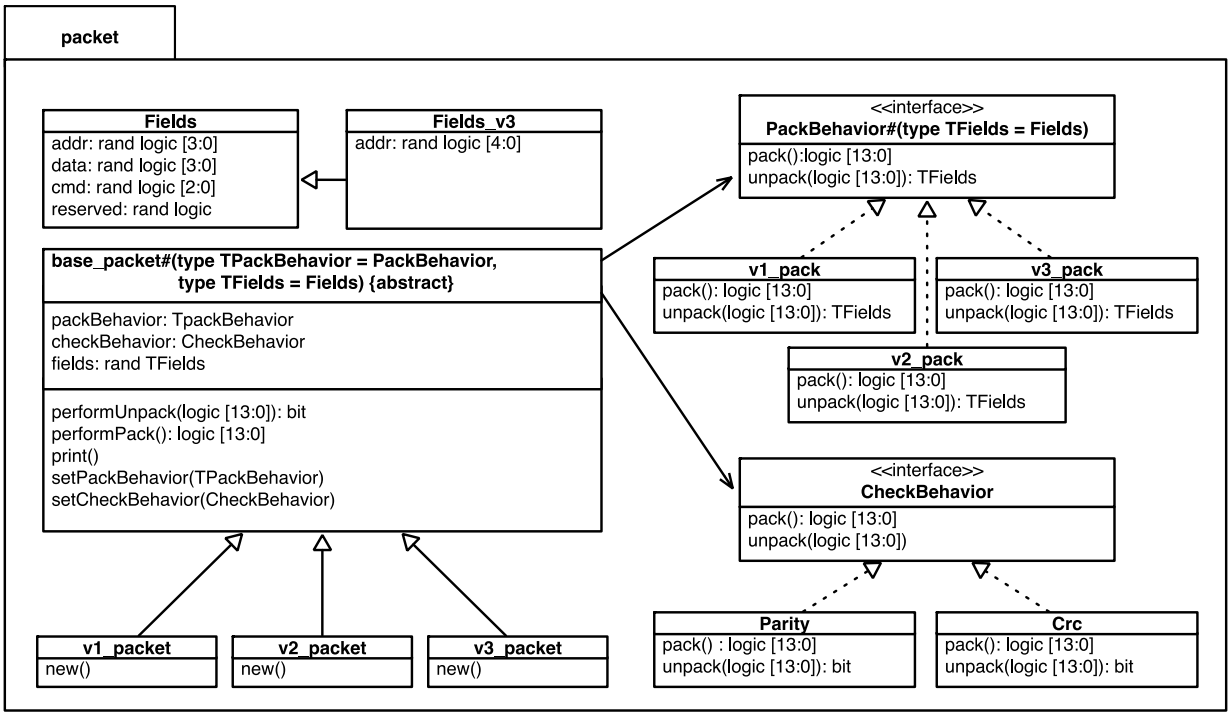

FIGURE 9. PACKET CLASS APPLICATION UML CLASS DIAGRAM OF STRATEGY PATTERN

The UML class diagram looks very similar to the Duck Simulator UML class diagram which is one of the major benefits of using Design Patterns. It becomes easier to describe complicated object relationships because there is a shared lexicon to describe class interactions. Replace "fly" and "quack" with "pack" and "unpack", and the diagram is structurally the same as the Duck Simulator.

This example becomes more complicated because of the use of parameterized classes and parameterized class interfaces based of the Field class used. If there is only one Field class, all of the parameterization can be stripped away. However, it is useful to show a potential next step for a Strategy Pattern implementation that is required to work with multiple types. Each packet class is a combination of Fields, a PackBehavior and a CheckBehavior. Each of these components is reusable and mixable with other objects or future packet types.

```
class v1_pack implements PackBehavior;
   virtual function Fields unpack(logic [13:0] raw);
      Fields fields    = new;
      fields.reserved  = raw[11];
      fields.addr      = raw[10:7];
      fields.data      = raw[6:3];
      fields.cmd       = raw[2:0];
      return fields;
   endfunction
```

FIGURE 10. STRATEGY PATTERN APPLICATION TO VERIFICATION V1_PACK CLASS BEHAVIOR [21]

```
class v1_packet extends base_packet;
   function new();
      v1_pack p = new();
      Crc c = new();

      setPackBehavior(p);
      setCheckBehavior(c);
   endfunction
endclass
```

FIGURE 11. STRATEGY PATTERN APPLICATION TO VERIFICATION V1_PACKET CLASS [21]

The next request for this problem might be to implement a new behavior that creates packets with a "cmd" field that is illegal when it is packed. This could lead to the creation of a new PackBehavior called "v1_pack_illegal_cmd" that could easily be swapped in and used. The Strategy Pattern allows for swapping out encapsulated algorithms because of the use of a defined interface.

V. THE DECORATOR PATTERN WITH STARBUZZ COFFEE

"Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extended functionality."

- Definition of the Decorator Pattern from "Design Patterns" [1, p. 175]

The Decorator Pattern allows for new responsibilities to be added to a specific object instance dynamically. While the Strategy Pattern dealt with replacing a single function at runtime, the Decorator Pattern can add multiple behaviors onto an object. The trick is to create a structure similar to a linked-list. The example described in "Head First Design Patterns" is of a coffee shop that sells different types of coffee and the optional add-ons to add to that coffee. An order could be a "Dark Roast with Double Mocha and Whip", which has a string name that would need to be printed on the receipt and also a cost that takes into account the coffee and its add-ons. The UML class diagram in Figure 12 describes the solution using the Decorator Pattern.

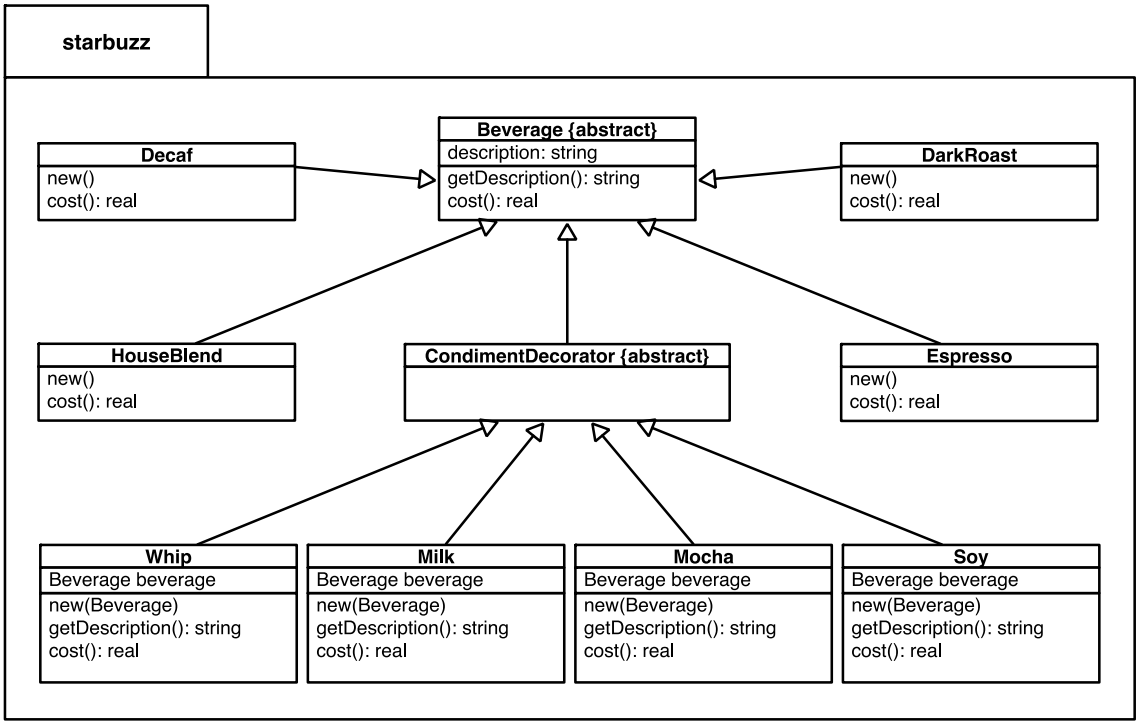

FIGURE 12. DECORATOR PATTERN UML CLASS DIAGRAM FOR STARBUZZ COFFEE

The use of a reference handle "Beverage" used in the decorator classes is the interesting part of the Decorator Pattern. The decorator classes in this example are the condiments, such as Whip and Milk. These decorators have a reference to a class Beverage, which is the abstract parent of all of these classes. Since "polymorphism allows the use of a variable of the superclass type to hold subclass objects and to reference the methods of those subclasses directly from the superclass variable" [2, p. 150], it allows the decorators to use a superclass handle that can reference the coffee classes, as well as fellow condiments classes.

```
class Mocha extends CondimentDecorator;

   Beverage beverage;

   function new(Beverage beverage);
      this.beverage = beverage;
   endfunction

   virtual function string getDescription();
      return {beverage.getDescription(), ", Mocha"};
   endfunction

   virtual function real cost();
      return (0.20 + beverage.cost());
   endfunction

endclass
```

FIGURE 13. DECORATOR PATTERN CLASS EXAMPLE IN SYSTEMVERILOG MOCHA.SV [8]

The Mocha condiment class in Figure 13 shows the method of how the Decorator Pattern achieves its chaining behavior. When "getDescription()" is called on this class, it calls "beverage.getDescription()", which would call its own "beverage.getDescription()" until it reached the coffee type. The constructor (in SystemVerilog this is known as the function "new") offers a convenient and clever location to provide the handle to the parent Beverage object.

```
module top;
   import starbuzz::*;

   Beverage beverage;
   string str;

   initial begin
      beverage  = Darkroast::new;
      beverage  = Mocha::new(beverage);
      beverage  = Mocha::new(beverage);
      beverage  = Whip::new(beverage);
      str.realtoa(beverage.cost());
      $display({beverage.getDescription(), " $", str});
```

FIGURE 14. DECORATOR PATTERN EXAMPLE USE WITH TYPED CONSTRUCTOR [8]

Figure 14 is an example of the ideal implementation of a Decorator Pattern in SystemVerilog using a new feature from SystemVerilog 1800-2012 called "Typed Constructor Calls" [2, p. 140]. This allows a class to return its object without needing an intermediary handle. An example of a "Typed Constructor Call" is "Darkroast::new". The condiments, such as "Mocha", pass to their constructor a handle to the Beverage object that it is encapsulating. In Figure 14 we implemented the order "Dark Roast with Double Mocha and Whip" and displayed the cost and the description to the screen of the encapsulated order.

Earlier versions of SystemVerilog did not have "Typed Constructor Calls". Figure 15 illustrates using the Decorator Pattern without "Typed Constructor Calls" which is painful to look at. It requires a number of temporary class handles, and a confusing pattern of creating a new instance and then passing that instance handle to the next constructor. This

language deficiency in previous versions of SystemVerilog might be the reason why the Decorator Pattern was seldom used, even though technically possible.

```
module top;
   import starbuzz::*;

   Beverage beverage;
   DarkRoast darkroast;
   Mocha mocha;
   Whip whip;
   string str;

   initial begin
      darkroast  = new;
      beverage   = new darkroast;
      mocha      = new(beverage);
      beverage   = new mocha;
      mocha      = new(beverage);
      beverage   = new mocha;
      whip       = new(beverage);
      beverage   = new whip;
      str.realtoa(beverage.cost());
      $display({beverage.getDescription(), " $", str});
```

FIGURE 15. DECORATOR PATTERN EXAMPLE USE WITHOUT TYPED CONSTRUCTOR [8]

DECORATOR PATTERN APPLIED TO VERIFICATION

At DVCon 2015 John Dickol published a paper titled "SystemVerilog Constraint Layering via Reusable Randomization Policy Classes" [22]. The paper provided a method to dynamically add SystemVerilog constraints to a randomized class, an incredibly powerful technique. At its core, the method proposed by Dickol involved setting up a queue of rand class objects that could be added to a class dynamically. An alternative to creating a queue of rand class objects is to use the Decorator Pattern to add this capability.

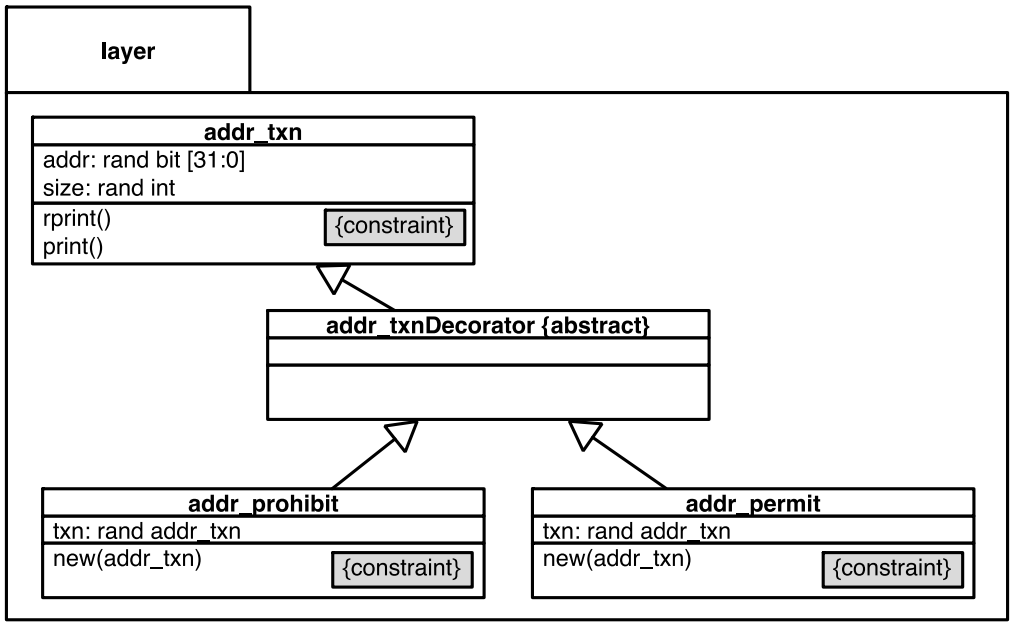

FIGURE 16. LAYERED CONSTRAINTS USING THE DECORATOR PATTERN UML CLASS DIAGRAM

The Decorator Pattern could be applied to the layered constraint problem using the UML class diagram from Figure 16. The UML class diagram has the addition of the "{constraint}" box to signify where constraints lie. We want to apply additively the constraints from "addr_prohibit" and "addr_permit" to our "addr_txn" class. The setup of this example could be compared to some of the early steps in the Dickol paper. We can make the analogy that applying new constraints to a class is similar to how the Decorator Pattern was used at the coffee shop example to add new condiments to a coffee.

```
class addr_permit extends addr_txnDecorator;
   rand addr_txn txn;

   function new(addr_txn txn);
      this.txn = txn;
   endfunction

   constraint c_addr_permit {
      addr inside {['h00000000 : 'h0000FFFF - txn.size]} ||
      addr inside {['h10000000 : 'h1FFFFFFF - txn.size]};

      txn.addr == addr;
      txn.size == size;
   }

endclass
```

FIGURE 17. DECORATOR PATTERN APPLIED TO LAYERED CONSTRAINTS ADDR_PERMIT [23]

The constructor function "new" in Figure 17 works just as the Decorator Pattern did in the coffee shop example, saving a reference to the object it was constructed with. The SystemVerilog constraints are similar to other constraints. The only twist to enable the Decorator Pattern with SystemVerilog constraints is to tie the local randomization of "addr" with "txn.addr" and "size" with "txn.size".

```
module top;

   layer::addr_txn txn;

   initial begin
      txn = new;
      txn = layer::addr_prohibit::new(txn);
      txn = layer::addr_permit::new(txn);
      txn.rprint();
```

FIGURE 18. DECORATOR PATTERN APPLIED TO LAYERED CONSTRAINTS EXAMPLE USAGE CALL [23]

Using the Decorator Pattern is intuitive and similar to how this pattern is applied in any programming language, taking an object and encapsulating it with a new behavior or new behaviors. The key to enabling the SystemVerilog constraints was adding a constraint to tie the randomized variable to the constructor's passed in handle. A benefit of this method is that the original txn class is untouched. New functionality is applied on top of a class using a decorator, instead of modifying the base class as was done in the Dickol method. By not altering the the original classes, we are applying the "Open/Closed principle" [24] of object-oriented programming, where we add new functionality, but do not alter the original implementation. The Dickol method was expanded upon in his paper, and provided features of layered constraints that would take consideration to determine if the Decorator Pattern by itself could implement.

VI. RESULTS

A major concern with using new SystemVerilog language features is compatibility with the simulators. Some of the SystemVerilog 1800-2012 features that enable Design Patterns are not fully supported in some of the simulators today. Figure 19 is a table of SystemVerilog features that are used in the Design Pattern examples provided and their compatibility with major simulators.

Simulator	local constructor	keyword "implements"	keyword "interface class"	typed constructor calls
A	Y	Y	Y	Y
B	Y	Y	Y	N
C	Y	N	N	N

FIGURE 19. SYSTEMVERILOG LANGUAGE FEATURES RELATED TO DESIGN PATTERNS VERSUS SIMULATOR COMPATABILITY

Five complete Design Pattern examples converted to SystemVerilog from the Java examples are available through this paper. Figure 20 provides the compatibility of each of those Design Pattern examples for each simulator. Each Design Pattern in the repository provided has a runfile for each simulator. A Verilog define directive of "PREFERRED" is given to demonstrate, if needed, the preferred implementation (P) versus a satisfactory implementation (S) depending on the simulator's compatibility.

Simulator	Decorator	Singleton	Observer	Strategy	State
A	P	P	P	P	P
B	S	P	P	P	P
C	S	P	N	N	N
P (Preferred Implementation) → S (Satisfactory Implementation) → N (Not Directly Supported)					

FIGURE 20. IMPLEMENTATION EFFECTIVENESS OF SELECTED DESIGN PATTERNS VERSUS SIMULATOR

VII. SUMMARY

This paper covered two patterns: the Strategy Pattern and the Decorator Pattern. Parallels between Java implementing the code examples and SystemVerilog were shown, and their full examples are available in the supporting code of this paper. There is a high fidelity between the Java implementation [18] and the SystemVerilog implementation [5] for all five attempted patterns, suggesting that SystemVerilog with the 1800-2012 release has finally achieved the language features it requires to implement something as ambitious as Design Patterns. Because Design Patterns themselves are some of the more complicated applications of object-oriented programming, this is a great feat, demonstrating how far SystemVerilog has come in its development.

The other patterns that were translated to SystemVerilog—Observer, State and Singleton—are also useful examples to show support for modern language features, arguably, necessary to properly implement Design Patterns. To show applications more familiar to a SystemVerilog user, two hardware verification applications of Design Patterns were explained. The Strategy Pattern was applied to the classical versioned packet protocol problem to show how different packing and unpacking functions could be swapped in through composition. Composition, which would in some form have been possible in prior versions of SystemVerilog, is now fully supported through the new keywords "class interface" and "implements" in SystemVerilog 1800-2012. The Decorator Pattern was applied to the previous year's DVCon paper by John Dickol, "SystemVerilog Constraint Layering via Reusable Randomization Policy Classes", to show an alternative way of layering constraints.

It is said that Design Patterns are not created, but are discovered. Object-oriented programming gives rise to these Design Patterns because certain types of problems can naturally be solved by applying some of these techniques. There is a verification framework in use today called UVM that has many ideas with names or functions very similar to what is described in "Design Patterns". The "UVM Factory" function is exactly the same idea as the "Factory Pattern" from "Design Patterns". The "UVM Analysis Port" has many similarities to the "Observer Pattern". It will be interesting to see what new UVM constructs can be created with these new language features.

A goal of this paper is to expose that SystemVerilog can now implement more modern software engineering ideas because of the inclusion of new language features. And, SystemVerilog can do so with a high fidelity when comparing the implementation details with a modern programming language such as Java. The comparison of the Java implementation to the SystemVerilog implementation is down to minor language choices, but the ideas and structure behind them are very close. At this time, in the UVM framework 1.2 there are no uses of the language constructs that were unique to SystemVerilog 1800-2012 and covered in this paper: "class interface", "implements" or "Typed Constructor Calls". That is for good reason – not all simulators support these language features at this time. The hope is that with concrete applications and example code, which this paper and its examples provide, these language features and the Design Patterns they enable can make their way into the object-oriented SystemVerilog designs yet to come.

VIII. Future Work

The application of the Decorator Pattern in SystemVerilog could have a number of uses. The paper demonstrated using the Decorator Pattern to add new constraints to a class. It could be useful to apply this same technique to add new functional coverage to an existing class as well.

Acknowledgement

A big thank you goes to O'Reilly Media [25] the publishers of "Head First Design Patterns" for allowing me to create translations of some of their Java examples to SystemVerilog to share in this paper. Also thank you to the co-writers of "Head First Design Patterns" Elisabeth Robson and Eric Freeman for writing such great examples for the book and posting them on Github. Thanks to Intel Technical Writer Myron Porter for technical writing expertise. Thanks to my employer Intel for sponsoring my attendance at DVCon to share this paper.

REFERENCES

[1] E. Gamma, R. Helm, R. Johnson and J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Indianapolis, IN: Addison-Wesley, 1994.

[2] IEEE Computer Society and the IEEE Standards Association Comporate Advisory Group, IEEE Standard for SystemVerilog 1800-2012, New York, NY: IEEE, 2013.

[3] E. Robson and E. Freeman, Head First Design Patterns, Sebastopol, CA: O'Reilly Media, Inc., 2004.

[4] E. Nelson, "Strategy Pattern from Head First Design Patterns in SystemVerilog," 31 12 2015. [Online]. Available: https://github.com/tenthousandfailures/systemverilog-design-patterns/tree/master/strategypattern/example. [Accessed 1 1 2016].

[5] E. Nelson, "The Strategy Pattern in SystemVerilog," 2014. [Online]. Available: http://tenthousandfailures.com/blog/2014/7/13/the-strategy-pattern-in-systemverilog.

[6] E. Nelson, "Observer Pattern from Head First Design Patterns in SystemVerilog," 31 12 2015. [Online]. Available: https://github.com/tenthousandfailures/systemverilog-design-patterns/tree/master/observer/example. [Accessed 1 1 2016].

[7] E. Nelson, "The Observer Pattern in SystemVerilog," 1 9 2014. [Online]. Available: http://tenthousandfailures.com/blog/2014/9/1/the-observer-pattern-in-systemverilog. [Accessed 1 1 2016].

[8] E. Nelson, "Decorator Pattern from Head First Design Patterns in SystemVerilog," 31 12 2015. [Online]. Available: https://github.com/tenthousandfailures/systemverilog-design-patterns/tree/master/decoratorpattern/example. [Accessed 1 1 2016].

[9] E. Nelson, "Singleton Pattern from Head First Design Patterns in SystemVerilog," 31 12 2015. [Online]. Available: https://github.com/tenthousandfailures/systemverilog-design-patterns/tree/master/singleton/example. [Accessed 1 1 2016].

[10] E. Nelson, "State Pattern from Head First Design Patterns in SystemVerilog," 31 12 2015. [Online]. Available: https://github.com/tenthousandfailures/systemverilog-design-patterns/tree/master/state/example. [Accessed 1 1 2016].

[11] E. Nelson, "Design Pattern SystemVerilog Code Examples on Github," 30 12 2015. [Online]. Available: https://github.com/tenthousandfailures/systemverilog-design-patterns. [Accessed 1 1 2016].

[12] Free Software Foundation, "GNU General Public Use License Version 2.0," 1991. [Online]. Available: http://www.gnu.org/licenses/old-licenses/gpl-2.0.en.html.

[13] E. Robson and E. Freeman, "Foundations of Programming: Design Patterns," Lynda.com, 13 12 2013. [Online]. Available: http://www.lynda.com/Developer-Programming-Foundations-tutorials/Foundations-Programming-Design-Patterns/. [Accessed 27 12 2015].

[14] Lynda.com, "Lynda.com," [Online]. Available: http://www.lynda.com. [Accessed 27 12 2015].

[15] Cadence, "Incisive Version 14.10," [Online]. Available: http://www.cadence.com.

[16] Mentor Graphics Corporation, "Questa Sim Version 10.4c," [Online]. Available: https://www.mentor.com.

[17] Synopsys, "VCS Version K-2015.09," [Online]. Available: http://www.synopsys.com.

[18] E. Robson and E. Freeman, "Stategy Pattern from Head First Design Patterns," 2014. [Online]. Available: https://github.com/bethrobson/Head-First-Design-Patterns/tree/master/src/headfirst/designpatterns/strategy. [Accessed 1 1 2016].

[19] H. Oliver, "Mixins over Inheritance," 8 11 2015. [Online]. Available: http://alisoftware.github.io/swift/protocol/2015/11/08/mixins-over-inheritance/. [Accessed 29 12 2015].

[20] M. Fowler, UML Distilled Third Edition: A Brief Guide to the Standard Object Modeling Language, 3rd Edition ed., Boston, MA: Addison-Wesley, 2006.

[21] E. Nelson, "The Strategy Pattern Applied to Verification," 30 12 2015. [Online]. Available: https://github.com/tenthousandfailures/systemverilog-design-patterns/tree/master/strategypattern/application. [Accessed 5 1 2016].

[22] J. Dickol, "SystemVerilog Constraint Layering via Reusable Randomization Policy Classes," in *Design and Verificaiton Conference*, San Jose, California, USA, 2015.

[23] E. Nelson, "Layered Constraints via the Decorator Pattern," 31 12 2015. [Online]. Available: https://github.com/tenthousandfailures/systemverilog-design-patterns/tree/master/decoratorpattern/application. [Accessed 1 1 2016].

[24] B. Meyer, Object-Oriented Software Construction, New York: Prentice Hall, 1988.

[25] O'Reilly Media, "O'Reilly Media," O'Reilly Media, 1 1 2016. [Online]. Available: http://www.oreilly.com. [Accessed 1 1 2016].