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

Eldon Nelson (eldon_nelson@ieee.org) Intel Corporation
Outline

• Design Patterns
  – Approach
• Composition versus Inheritance
• Strategy Pattern
  – Class Interface and Implements
  – Application to Evolving Packet Class
• Decorator Pattern
  – Typed Constructor Calls
  – Application to Layered Constraints
Full Code Examples

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

```systemverilog
class MiniDuckSimulator;
  static function void main();
  MallardDuck mallard = new();
  RubberDuck rubberDuckie = new();
```
<table>
<thead>
<tr>
<th>Duck {abstract}</th>
</tr>
</thead>
<tbody>
<tr>
<td>display()</td>
</tr>
<tr>
<td>QuackBehavior()</td>
</tr>
<tr>
<td>FlyBehavior()</td>
</tr>
</tbody>
</table>
Duck {abstract}

display()
QuackBehavior()
FlyBehavior()

MallardDuck

display()
quack()
fly()
```plaintext
Duck {abstract}
- display()
- QuackBehavior()
- FlyBehavior()

MallardDuck
- display()
- quack()
- fly()

RedheadDuck
- display()

RubberDuck
- display()
- squeak()
- flynoway()
```
Duck {abstract}
- display()
- QuackBehavior()
- FlyBehavior()

MallardDuck
- display()
- quack()
- fly()

RedheadDuck
- display()

RubberDuck
- display()
- squeak()
- flynoway()

DecoyDuck
- display()
- mutequack()
Duck {abstract}

- display()
- QuackBehavior()
- FlyBehavior()

- MallardDuck
  - display()
  - quack()
  - fly()

- RedheadDuck
  - display()

- RubberDuck
  - display()
  - squeak()
  - flynoway()

- DecoyDuck
  - display()
  - mutequack()
DroneDuck!
Duck {abstract}

display()
QuackBehavior()
FlyBehavior()

fly
quack

no fly
no quack

MallardDuck

display()
quack()
fly()

RubberDuck

display()
squeak()
flynoway()

RedheadDuck

display()

DecoyDuck

display()mutequack()

DroneDuck

display()
fly()}
Design Patterns 1994

• Is the definitive first text
• Examples in the programming language Smalltalk or in C++ with features from 1994
• Code fragment examples center around a Word Processor GUI Application
Eldon Nelson - Intel Corporation
Head First Design Patterns

- Examples written solely in Java
- Complete examples given for every exercise
- Examples are on a variety of whimsical fictional applications
Outline

• Design Patterns
  – Approach

• Composition versus Inheritance

• Strategy Pattern
  – Class Interface and Implements
  – Application to Evolving Packet Class

• Decorator Pattern
  – Typed Constructor Calls
  – Application to Layered Constraints
“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’”

- “Design Patterns” p.19
One Direction

Love Song

Santa Clara High School

Money
One Direction

Love Song

London High School

Money
One Direction

Arguments

Love Song

London High School

Function

Return

Money
One Direction

 arguments

 Function

 Return

 DVCon Lecture

 London High School

 Riot
Interface Class (Contract)

• “interface class” cannot define implementation
  – In contrast with an “virtual class” which can
  – Defines the arguments and return values

• “interface class” must have only “pure virtual function”

• No: constraints or covergroups
<<interface>>
QuackBehavior
quack()
**Interface Class (Contract)**

**JAVA**

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

**SYSTEMVERILOG**

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

```java
public class Squeak implements QuackBehavior {
    public void quack() {
        System.out.println("Squeak");
    }
}
```

SYSTEMVERILOG

```verilog
class Squeak implements QuackBehavior;
    virtual function void quack();
    $display("Squeak");
endfunction
endclass
```
• “In other words, don't check whether it IS-a duck: check whether it QUACKS-like-a duck, WALKS-like-a duck, etc, etc, depending on exactly what subset of duck-like behaviour you need to play your language-games with.”

– Alexis Martelli, 7/26/2000 in comp.lang.python
Outline

• Design Patterns
  – Approach

• Composition versus Inheritance

• **Strategy Pattern**
  – Class Interface and Implements
  – Application to Evolving Packet Class

• Decorator Pattern
  – Typed Constructor Calls
  – Application to Layered Constraints
Strategy Pattern

“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]
Duck {abstract}
flyBehavior: FlyBehavior
quackBehavior: QuackBehavior

display()
performQuack()
performFly()
setFlyBehavior()
setQuackBehavior()
Duck {abstract}
flyBehavior: FlyBehavior
quackBehavior: QuackBehavior

display()
performQuack()
performFly()
setFlyBehavior()
setQuackBehavior()

MallardDuck
new()
display()

RedheadDuck
new()
display()

RubberDuck
new()
display()

DecoyDuck
new()
display()
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

display()
performQuack()
performFly()
setFlyBehavior()
setQuackBehavior()

Duck {abstract}
  flyBehavior: FlyBehavior
  quackBehavior: QuackBehavior

  display()
  performQuack()
  performFly()
  setFlyBehavior()
  setQuackBehavior()

MallardDuck
  new()
  display()

RedheadDuck
  new()
  display()

RubberDuck
  new()
  display()
virtual class Duck;
    FlyBehavior flyBehavior;
    QuackBehavior quackBehavior;

function void setQuackBehavior(QuackBehavior qb);
    quackBehavior = qb;
endfunction

function void performQuack();
    quackBehavior.quack();
endfunction

Duck {abstract}
    flyBehavior: FlyBehavior
    quackBehavior: QuackBehavior

    display()
    performQuack()
    performFly()
    setFlyBehavior()
    setQuackBehavior()

MallardDuck
    new()
    display()

RedheadDuck
    new()
    display()

RubberDuck
    new()
    display()

DecoyDuck

Eldon Nelson - Intel Corporation
Favor Composition

“Favor object composition over class inheritance.”

- Second Principle of Object-Oriented Programming from “Design Patterns” [1, p. 20]
Strategy Pattern
Applied to Packet Problem

• Encapsulation of a changing packet format
• Fields change width and location
• Changeable checking algorithm – CRC or Parity
Policy Classes
AKA Strategy Pattern

Summary

Policy Classes

Each of UVM’s policy classes perform a specific task for uvm_object-based objects: printing, comparing, recording, packing, and unpacking.
Outline

• Design Patterns
  – Approach
• Composition versus Inheritance
• Strategy Pattern
  – Class Interface and Implements
  – Application to Evolving Packet Class
• Decorator Pattern
  – Typed Constructor Calls
  – Application to Layered Constraints
Decorator Pattern

“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]
Decorator Pattern
Example Starbuzz Coffee

- Coffee type like: HouseBlend or DarkRoast
- Has Condiments like: Mocha or Whip
- Each Coffee type has a price and each Condiment has a price
- An order could be
  - “Dark Roast with Double Mocha and Whip”
  - Give the string and price for this order
CondimentDecorator {abstract}

Milk
Beverage beverage
new(Beverage) 
getDescription(): string
cost(): real

Mocha
Beverage beverage
new(Beverage) 
getDescribe(): string
cost(): real

Beverage {abstract}
description: string

CondimentDecorator {abstract}

Milk
Beverage beverage
new(Beverage) 
getDescribe(): string
cost(): real

Mocha
Beverage beverage
new(Beverage) 
getDescribe(): string
cost(): real

Beverage {abstract}
description: string

CondimentDecorator {abstract}
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

dendcode
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

dendcode
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});

“Dark Roast Mocha Mocha Whip $1.65”
Decorator Pattern Application

• “SystemVerilog Constraint Layering via Reusable Randomization Policy Classes”
  John Dickol, DVCon 2015

• Dynamically applying multiple combinations of SystemVerilog constraints at runtime
layer

\begin{tabular}{|l|}
\hline
addr\_txn \\
\hline
\begin{tabular}{l}
addr: rand bit [31:0] \\
size: rand int \\
rprint() \\
print()
\end{tabular}
\hline
\end{tabular}

\begin{tabular}{|l|}
\hline
addr\_txnDecorator \{abstract\} \\
\hline
\end{tabular}

\begin{tabular}{|l|}
\hline
addr\_prohibit \\
\hline
\begin{tabular}{l}
txn: rand addr\_txn \\
new(addr\_txn)
\end{tabular}
\hline
\end{tabular}

\begin{tabular}{|l|}
\hline
addr\_permit \\
\hline
\begin{tabular}{l}
txn: rand addr\_txn \\
new(addr\_txn)
\end{tabular}
\hline
\end{tabular}

\{constraint\}
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
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
module top;

layer::addr_txn txn;

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

Decorator Pattern
Layered Constraint
Awesome Let's Use It!

- Software Design Pattern ideas are widely accepted and refined over decades
- The examples ported to SystemVerilog have a high fidelity with their Java sources
  - Meaning SystemVerilog has feature parity to meet these examples without compromises with SV 2012
- Simulator Vendor Compatibility
### SV 2012

#### Simulator Compatibility

<table>
<thead>
<tr>
<th>Simulator</th>
<th>local constructor</th>
<th>keyword “implements”</th>
<th>keyword “interface class”</th>
<th>typed constructor calls</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>B</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>C</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
</tbody>
</table>

**New SV 1800-2012**

- New SV 1800-2012
- New SV 1800-2012
- New SV 1800-2012
## Design Pattern Compatibility per Simulator

<table>
<thead>
<tr>
<th>Simulator</th>
<th>Decorator</th>
<th>Singleton</th>
<th>Observer</th>
<th>Strategy</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>P</td>
<td>P</td>
<td>P</td>
<td>P</td>
<td>P</td>
</tr>
<tr>
<td>B</td>
<td>S</td>
<td>P</td>
<td>P</td>
<td>P</td>
<td>P</td>
</tr>
<tr>
<td>C</td>
<td>S</td>
<td>P</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
</tbody>
</table>

- **P** (Preferred Implementation) ➔ **S** (Satisfactory Implementation) ➔ **N** (Not Directly Supported)
Outline

• Design Patterns
  – Approach
• Composition versus Inheritance
• Strategy Pattern
  – Class Interface and Implements
  – Application to Evolving Packet Class
• Decorator Pattern
  – Typed Constructor Calls
  – Application to Layered Constraints
Opportunities with Design Patterns

• Excited to see how these new language features give an alternative to inheritance
• Excited to see more use of the Decorator Pattern as it is one of the most common
  – constraints are just the beginning
• Vendor support will come
  – but please tell them you want it!
Questions

Please Vote at
http://vote.dvcon.org
1. Why use an “interface class” when you could use an “virtual class” and put in your own “pure virtual functions”?

2. Show me the horror that is the Decorator Pattern without “typed constructor calls”

3. What can I do to improve the adoption of these SystemVerilog features in the simulators?

4. Is the paper better than your presentation – cause… you know?
1. Why do you sign your @ieee.org email address on all your papers?

2. What pattern if you had more time would you like to share next? Well, other than the Packet Protocol protocol one you said you skipped.
uvm_default_printer

uvm_printer uvm_default_printer = uvm_default_table_printer

The default printer policy. Used when calls to uvm_object::print or uvm_object::sprint do not specify a printer policy.

The default printer may be set to any legal uvm_printer derived type, including the global line, tree, and table printers described above.
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});

“Dark Roast Mocha Mocha Whip $1.65”
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});
Strategy Pattern

• Removed from Presentation for Time Issues
packet

Fields
- addr: rand logic [3:0]
- data: rand logic [3:0]
- cmd: rand logic [2:0]
- reserved: rand logic

Fields_v3
- addr: rand logic [4:0]

base_packet#: (type TPackBehavior = PackBehavior,
              type TFields = Fields) {abstract}
- packBehavior: TpackBehavior
- checkBehavior: CheckBehavior
- fields: rand TFields

performUnpack(logic [13:0]): bit
performPack(): logic [13:0]
print()
setPackBehavior(TPackBehavior)
setCheckBehavior(CheckBehavior)

new()

<<interface>>
PackBehavior#: (type TFields = Fields)
- pack(): logic [13:0]
- unpack(logic [13:0]): TFields

v1_pack
- pack(): logic [13:0]
- unpack(logic [13:0]): TFields

v2_pack
- pack(): logic [13:0]
- unpack(logic [13:0]): TFields

v3_pack
- pack(): logic [13:0]
- unpack(logic [13:0]): TFields

<<interface>>
CheckBehavior
- pack(): logic [13:0]
- unpack(logic [13:0])

Parity
- pack(): logic [13:0]
- unpack(logic [13:0]): bit

Crc
- pack(): logic [13:0]
- unpack(logic [13:0]): bit
pack(): logic [13:0]
unpack(logic [13:0]): TFields

v1_pack.pack(): logic [13:0]
v1_pack.unpack(logic [13:0]): TFields

v2_pack.pack(): logic [13:0]
v2_pack.unpack(logic [13:0]): bit

Parity.pack(): logic [13:0]
Parity.unpack(logic [13:0]): bit

Crc.pack(): logic [13:0]
Crc.unpack(logic [13:0]): TFields

v3_pack.pack(): logic [13:0]
v3_pack.unpack(logic [13:0]): bit

Fields
addr: rand logic [3:0]
data: rand logic [3:0]
cmd: rand logic [2:0]
reserved: rand logic

Fields_v3
addr: rand logic [4:0]

base_packet#(type TPackBehavior = PackBehavior,
type TFields = Fields) {abstract}

packBehavior: TPackBehavior
checkBehavior: CheckBehavior
fields: rand TFields

performUnpack(logic [13:0]): bit
performPack(): logic [13:0]
print()
<<interface>>

PackBehavior#(type TFields = Fields)

pack(): logic [13:0]
unpack(logic [13:0]): TFields

v1_pack

pack(): logic [13:0]
unpack(logic [13:0]): TFields

v2_pack

pack(): logic [13:0]
unpack(logic [13:0]): TFields

v3_pack

pack(): logic [13:0]
unpack(logic [13:0]): TFields
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
class v1_packet extends base_packet;
  function new();
    v1_pack p = new();
    Crc c = new();
  setPackBehavior(p);
  setCheckBehavior(c);
endfunction
endclass