# SystemVerilog Virtual Classes, Methods, Interfaces

## Their Use in Verification and UVM


Clifford E. Cummings
Sunburst Design, Inc.

Heath Chambers
HMC Design Verification


**World-class SystemVerilog & UVM Verification Training**

**Life is too short for bad or boring training!**

March 21-22, 2018
Silicon Valley

# What's in the Paper? … And Our Agenda

Classes & virtual classes
Methods & virtual methods
Extended & derivative classes

Upcasting & downcasting
Upcasting & downcasting in UVM

Pure virtual methods

Virtual interfaces
Virtual interface usage in UVM

Summary & Conclusions

**More examples and more detail in the paper**

# Virtual Classes & Methods

## A Brief Introduction



**Guideline #1:** Class methods should be `function` and `void function`. Only use `task` when the method consumes simulation time

- **`virtual`** classes - cannot be constructed

**`virtual`** class handles can be declared and referenced but cannot be **`new()`**- ed

- **`virtual`** classes can have non-virtual, **`virtual`** & **`pure virtual`** methods

Virtual class methods are non-virtual by default

Only virtual classes can declare **`pure virtual`** methods

- Non-virtual classes can only have non-virtual & **`virtual`** methods

Fixed during construction and cannot be changed during simulation

- Non-virtual methods are set when a class object is constructed
- **`virtual`** methods can be modified at run-time

Accomplished by assigning a derivative object handle to a base object handle

**Guideline #2:** Declare SystemVerilog class methods to be `virtual` methods unless there is a very good reason to prohibit method polymorphism

# Assigning Extended & Base Class Handles

## Upcasting & Downcasting

# Upcasting & Downcasting

## Assuming Compatible Class Types

"***Upcasting*** is casting to a ***supertype***, while ***downcasting*** is casting to a ***subtype***.
*Upcasting is always allowed, but downcasting involves a type check*
… and can throw a **`ClassCastException`**"

**From "stackoverflow" reference in the paper**

- To paraphrase:
  - Any extended or derivative class handle can be assigned to a base class handle
  - Only some base class handles can be assigned to an extended class handle

**SystemVerilog requires a type-check for *downcasting***

**We do upcasting and downcasting very frequently in UVM !**
***(examples will be shown)***

# Assigning Extended & Base Class Handles

## Package with Base & Extended Classes

```
package test_classes;
  class base;
    bit [7:0] a;

    function void seta(bit [7:0] din);
      a = din;
    endfunction

    function bit [7:0] geta();
      return(a);
    endfunction
  endclass

  class ext extends base;
    bit [7:0] data;

    function void setdata(bit [7:0] din);
      data = din;
    endfunction

    function bit [7:0] getdata();
      return(data);
    endfunction
  endclass
endpackage
```

**base class**

```
function void showit;
  $display("BASE(%m): a=%2d", a);
endfunction
```

**ext extended class**

**Inherits:**
- **a variable**
- **seta() method**
- **geta() method**

```
function void showit;
  $display(" EXT(%m): data=%2d   a=%2d",
                      data, a);
endfunction
```

**What assignments are legal between base & extended class handles?**

**Example assignments on the next slides**

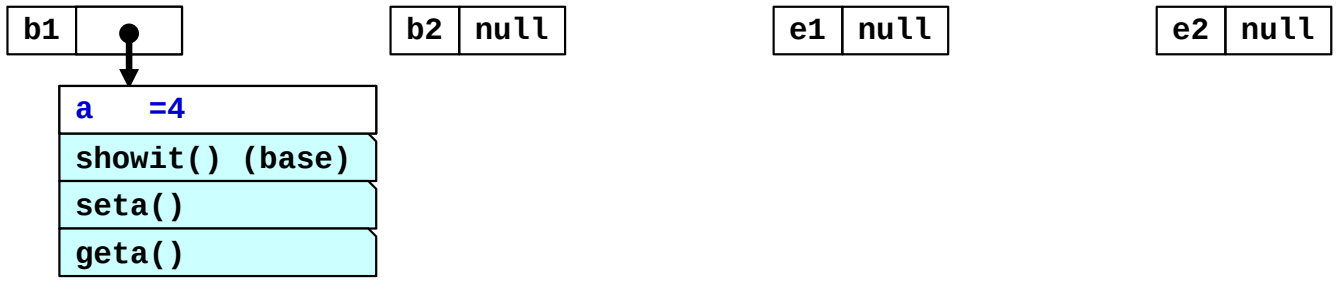
# Construct A Base Class Object

**Handle and bit declarations**

```
module test;
  ...
  base b1, b2;
  ext  e1, e2;
  bit  e1good,
       e2good;
```

```
...
initial begin
  b1 = new();
  b1.seta(4);
  b1.showit();
  ...
```

- Construct the `b1` class object
- Set the `b1` variable `a=4`
- Show the `b1.a` variable

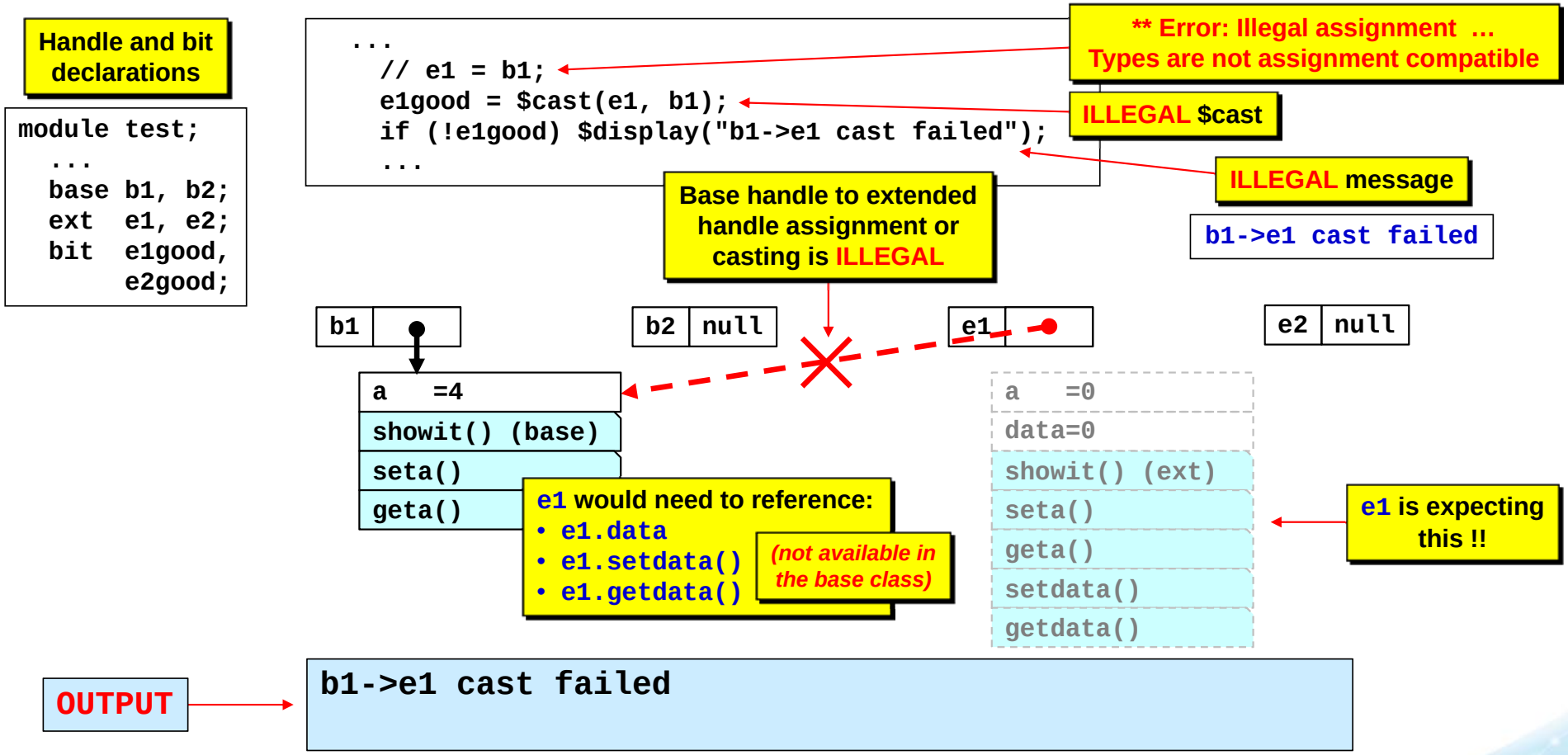



**OUTPUT**

```
BASE(test_classes.base.showit): a= 4
```

# Illegal Casting of Base to Extended Handle

## Illegal Downcasting



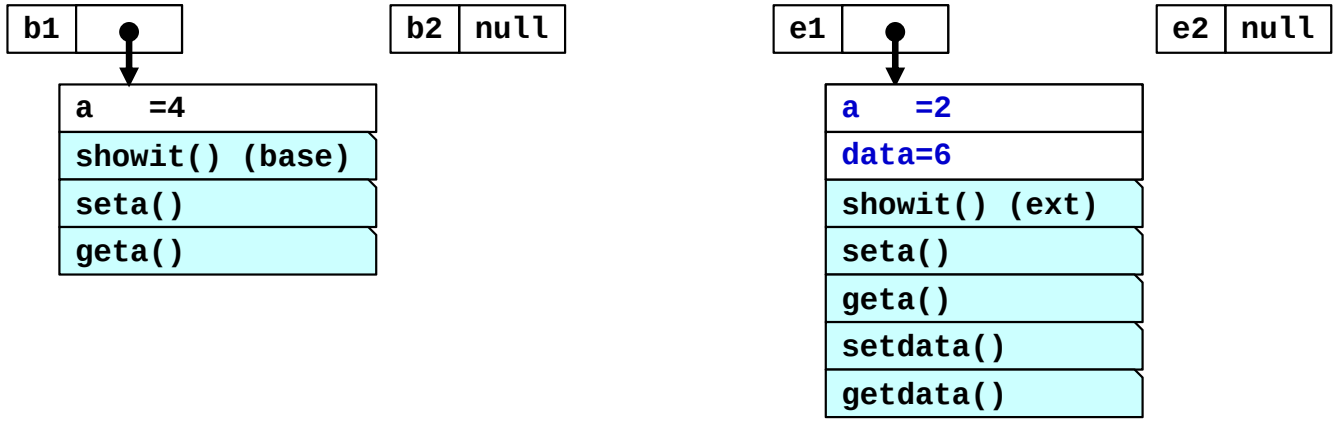
# Construct An Extended Class Object

**Handle and bit declarations**

```
module test;
  ...
  base b1, b2;
  ext  e1, e2;
  bit  e1good,
       e2good;
```

```
...
e1 = new();
e1.seta(2);
e1.setdata(6);
e1.showit();
...
```

- Construct the `e1` class object
- Set the `e1` variable `a=2`
- Set the `e1` variable `data=6`
- Show the `e1.a` & `e1.data` variables




**OUTPUT**

```
EXT(test_classes.ext.showit): data= 6   a= 2
```

# Legal Assignment of Extended to Base Handle

## Legal Upcasting

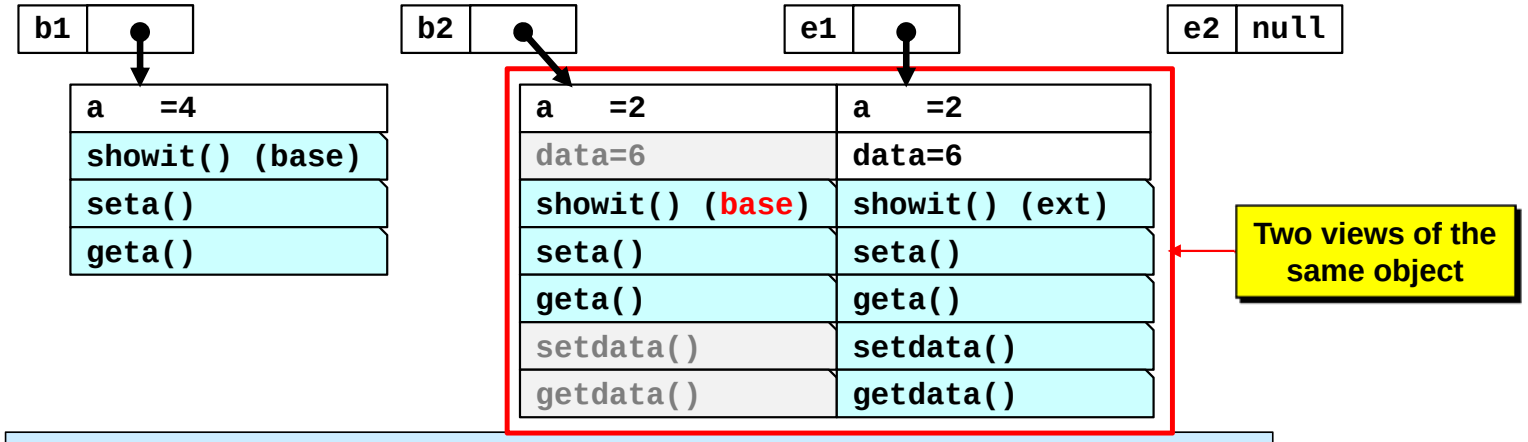
**Handle and bit declarations**

```
module test;
  ...
  base b1, b2;
  ext  e1, e2;
  bit  e1good,
       e2good;
```

```
  ...
b2 = e1;
b2.showit();
...
```

**Extended handle to base handle assignment is LEGAL**

**Execute base class `showit()` method**




**Two views of the same object**

**OUTPUT**

```
BASE(test_classes.base.showit): a= 2
```

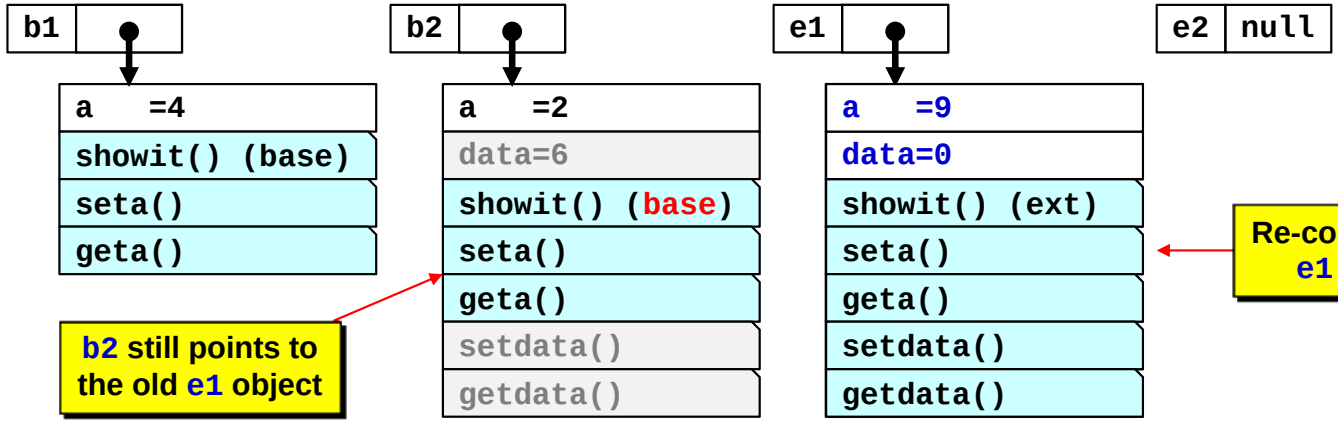
# Re-Construct The Extended Class Object

**Handle and bit declarations**

```
module test;
  ...
  base b1, b2;
  ext  e1, e2;
  bit  e1good,
       e2good;
```

```
  ...
b2 = e1;
b2.showit();

e1 = new();
e1.seta(9);
e1.showit();
...
```

- Re-construct the `e1` class object
- Set the `e1` variable `a=9`
- The `e1` variable keeps `data=0`
- Show the `e1.a` & `e1.data` variables




**OUTPUT**

```
EXT(test_classes.ext.showit): data= 0   a= 9
```

# Illegal Access to Extended Object Methods

## From a Base Class Handle

**Handle and bit declarations**

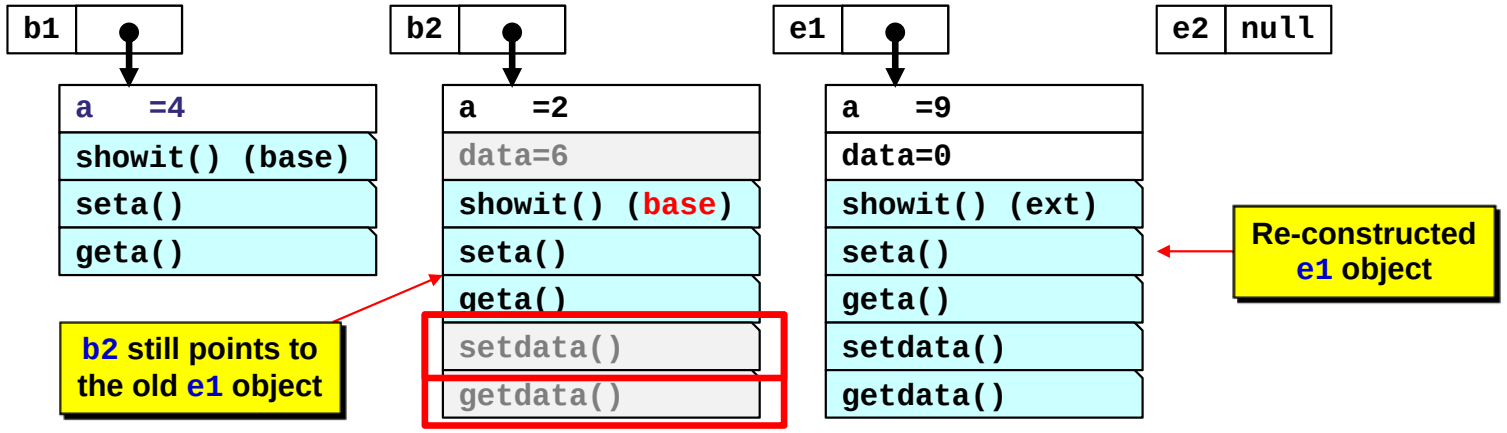
```
module test;
  ...
  base b1, b2;
  ext  e1, e2;
  bit  e1good,
       e2good;
```

```
    ...
    // b2.setdata(9);
    // m_data = b2.getdata();
    ...
```

**NOTE: b2 handle does not have access to setdata() or getdata() methods**

** Error: Field/method name (setdata) not in 'b2'

** Error: Field/method name (getdata) not in 'b2



# Casting-Back Base to Extended Handle

## Downcasting Requires a Type-Check

**Handle and bit declarations**

```
module test;
  ...
  base b1, b2;
  ext  e1, e2;
  bit  e1good,
       e2good;
```

```
    ...
    // e2 = b2;
    e2good = $cast(e2, b2);
    if (e2good) $display("b2->e2 cast PASSED!");
    e2.showit;
  end
endmodule
```

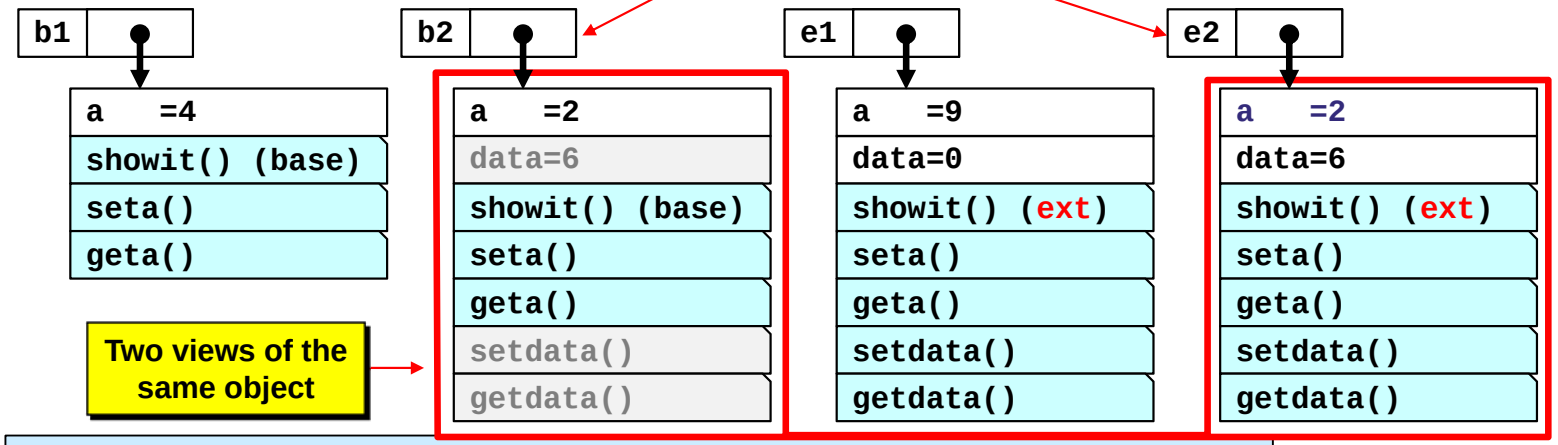
**** Error: Illegal assignment ... Types are not assignment compatible**

**LEGAL $cast**

**LEGAL message**

```
b2->e2 cast PASSED
```

**b2 and e2 point to the same object**




**Two views of the same object**

**OUTPUT**

```
b2->e2 cast PASSED!
 EXT(test_classes.ext.showit): data= 6   a= 2
```

# Full Handle Assignment Example

```
module test;
  ...
 initial begin
    b1 = new();
    b1.seta(4);
    b1.showit();
    // e1 = b1;
    e1good = $cast(e1, b1); // ILLEGAL $cast
    if (!e1good) $display("b1->e1 cast failed");
    e1 = new();
    e1.seta(2);
    e1.setdata(6);
    e1.showit();
    b2 = e1;
    b2.showit();
    e1 = new();
    e1.seta(9);
    e1.showit();
    // b2.setdata(9);
    // m_data = b2.getdata();
    // e2 = b2;
    e2good = $cast(e2, b2); // LEGAL $cast
    if (e2good) $display("b2->e2 cast PASSED!");
    e2.showit;
  end
endmodule
```

Construct base object

Commented code caused compiler errors

ILLEGAL base class to extended class `$cast` operation

Construct extended object

Copy extended handle to base handle

LEGAL base class to extended class `$cast` operation

# Base & Extended Class Handle Assignments

## Assuming Compatible Class Types

- Any extended handle can be assigned to a base handle
  - Can be a direct assignment or a **`$cast`**-assignment
  - Base handles cannot access *extended* methods and data

- Some base handles can be assigned to extended handles
  - Requirements:
    - (1) Extended object had to be constructed
    - (2) Constructed extended-handle had to be assigned to a base handle
    - (3) Base handle must be **`$cast`**-back to an extended handle

- This technique allows creation of single base-class reference table
  - Any extended handle can be stored in the table
  - Any stored handle can be restored (**`$cast`**) to an extended handle

**This is why we care !!**

**Upcasting**

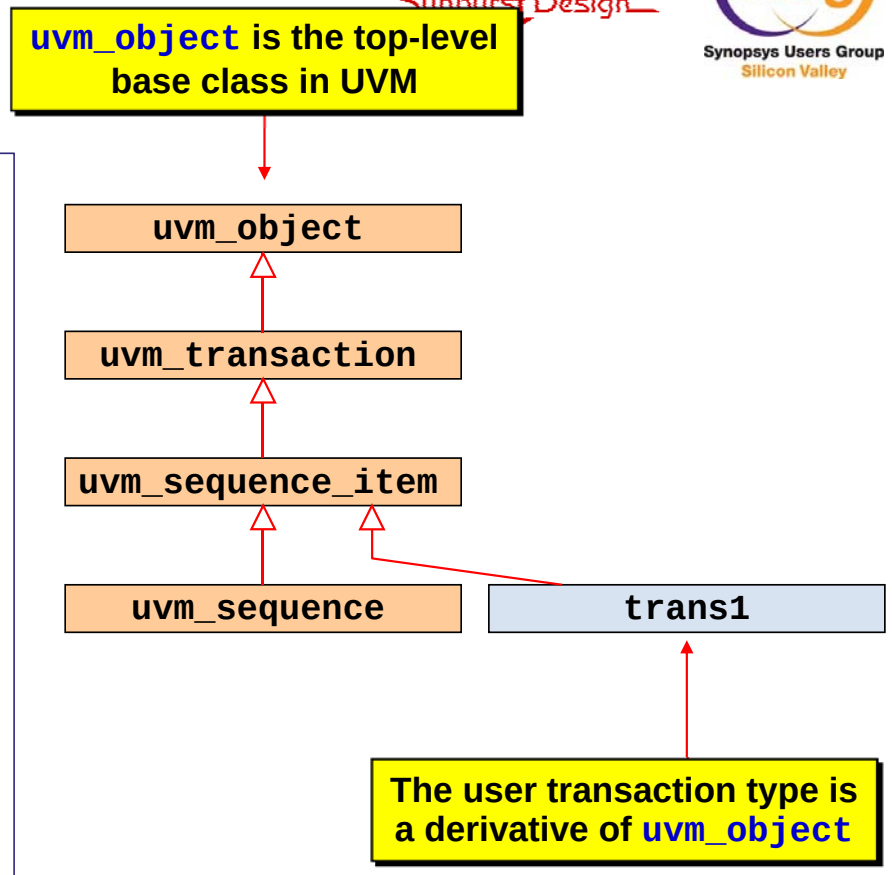
**Downcasting**

**Used by OVM & UVM**

# User-Defined Transaction Class

## Derivative of uvm_object

```
class trans1 extends uvm_sequence_item;
  `uvm_object_utils(trans1)

       logic [15:0] dout;
  rand bit   [15:0] din;
  rand bit          ld, inc, rst_n;
  ...
  function void do_copy(uvm_object rhs);
    trans1 tr;
    if(!$cast(tr, rhs))
      `uvm_fatal("trans1", "FAIL: do_copy() cast");
    super.do_copy(rhs);
    dout  = tr.dout;
    din   = tr.din;
    ld    = tr.ld;
    inc   = tr.inc;
    rst_n = tr.rst_n;
  endfunction
  ...
endclass
```



# Transaction Class do_copy() Method

## Example Usage from Scoreboard Predictor

```
class trans1 extends uvm_sequence_item;
  `uvm_object_utils(trans1)

       logic [15:0] dout;
  rand bit   [15:0] din;
  rand bit          ld, inc, rst_n;
  ...
  function void do_copy(uvm_object rhs);
    trans1 tr;
    if(!$cast(tr, rhs))
      `uvm_fatal("trans1", "...");
    super.do_copy(rhs);
    dout  = tr.dout;
    din   = tr.din;
    ld    = tr.ld;
    inc   = tr.inc;
    rst_n = tr.rst_n;
  endfunction
  ...
endclass
```

The scoreboard predictor has `sb_calc_exp()` function

The `sb_calc_exp()` function is called with `trans1 t` handle

```
function trans1 ... sb_calc_exp (trans1 t);
  ...
  trans1 tr = trans1::type_id::create("tr");
  ...
  tr.copy(t);
  ...
  return(tr);
endfunction
```

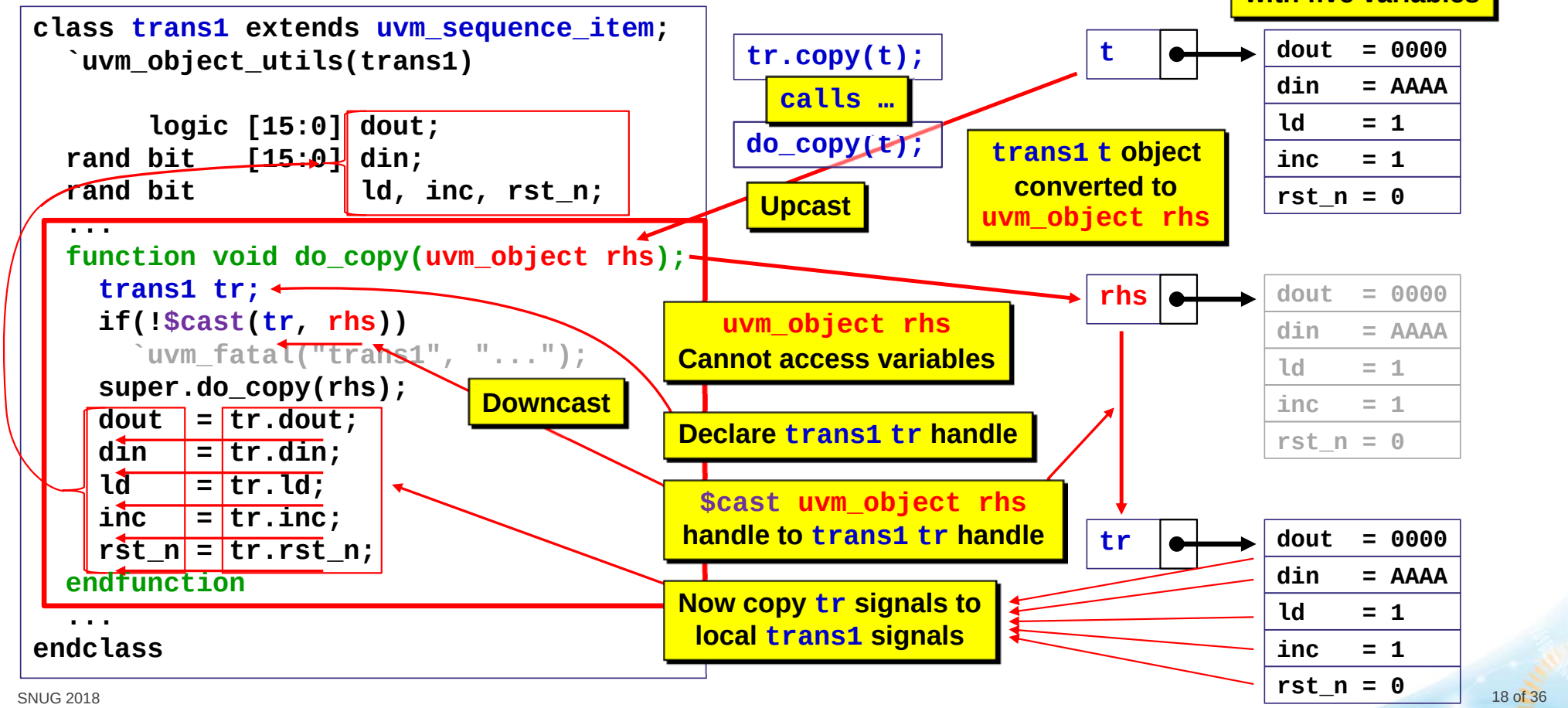
Local `trans1 tr` object is created

All fields of the `t` object are copied to the fields of the local `tr` object

The `sb_calc_exp()` function returns the `tr` handle

# Transaction Class do_copy() Method

## Upcasting & Downcasting



# Upcasting & Downcasting Variable Names

## Avoid Confusing Names



Previous slide - We named the local `trans1` handle `tr`

- Many industry examples name the local transaction handle `rhs_`
- Using `rhs_` means that casting is done in the form `$cast(rhs_, rhs);`

This is confusing and therefore a poor practice

- Causes fields to be referenced as `rhs_.field_1` , …

Easy to confuse the `uvm_object` `rhs` handle with the transaction class `rhs_` handle

- Better practice: Use a transaction handle name like `tr`

Or another name that is visually distinct

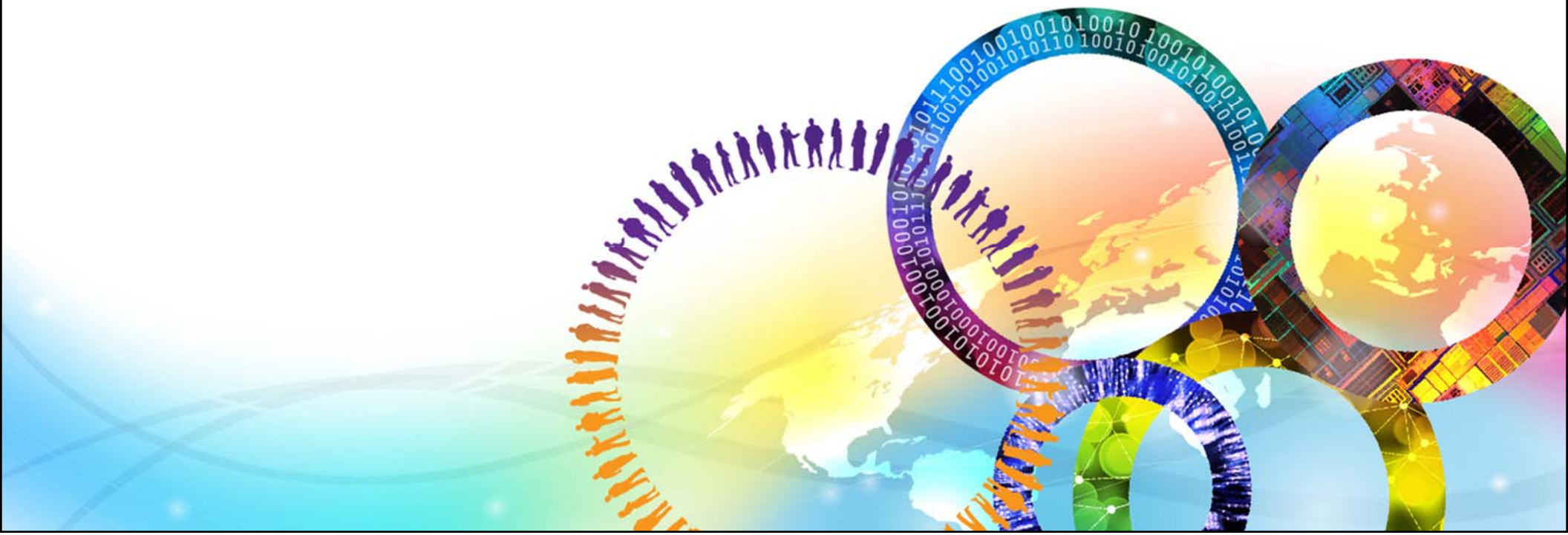
**Guideline #3:** Declare local transaction handles using distinct names such as `tr` and avoid local transaction handle names such as `rhs_`

Sunburst Design

snug
Synopsys Users Group
Silicon Valley

# Pure Virtual Methods

## SystemVerilog-2009 Enhancement



# Pure Virtual Methods

## Two important purposes

```
virtual class vc1a;
  bit [7:0] a;

  pure virtual function void seta(bit [7:0] val);
endclass
```

**pure virtual method**

**(1) pure virtual methods can only be a method prototype**

**No method body allowed**

**No endfunction / endtask allowed**

```
class ex1a extends vc1a;

  virtual function void seta(bit [7:0] val);
    a = val;
  endfunction
endclass
```

**ex1a *MUST* override seta()**
***(must provide an implementation)***

**(2) pure virtual methods *must* be overridden in a non-virtual class**

**NOTE: pure keyword is only legal in virtual classes**

# Pure Virtual Methods



**`virtual` classes**

```
virtual class vc1a;
  bit [7:0] a;

  pure virtual function void seta(bit [7:0] val);
endclass
```

**`pure virtual` method**

```
virtual class vc1b;
  bit [7:0] a;

  pure virtual function void seta(bit [7:0] val);
endclass
```

**`pure virtual` method**

```
virtual class vc2a extends vc1a;


endclass
```

**vc2a *does NOT* override `seta()` method**

```
virtual class vc2b extends vc1b;

   virtual function void seta(bit [7:0] val);
    a = val;
  endfunction
endclass
```

**vc2b *DOES* override `seta()` method**

***non*-`virtual` classes**

```
class ex1a extends vc2a;

  virtual function void seta(bit [7:0] val);
    a = val;
  endfunction
endclass
```

**ex1a *MUST* override `seta()`**
***(must provide an implementation)***

```
class ex1b extends vc2b;

  // optional override of seta()
endclass
```

**ex1b can *OPTIONALLY* override `seta()`**

# UVM Pure Virtual Method Example

## uvm_subscriber - pure write() Method

```
virtual class uvm_subscriber #(type T=int) extends uvm_component;
  ...

  pure virtual function void write(T t);
endclass
```

`uvm_subscriber` `virtual` class

`pure` keyword is only legal in `virtual` classes

The class parameter `type` (`T`) is used in this method prototype

No `endfunction` keyword allowed!

```
class tb_cover extends uvm_subscriber #(trans1);
  ...
  function void write (trans1 t);

    dout  = t.dout;
    din   = t.din;
    ld    = t.ld;
    inc   = t.inc;
    rst_n = t.rst_n;
    `uvm_info("tb_cover", "Taking covergroup sample ...", UVM_HIGH)
    cg.sample();
  endfunction
endclass
```

User-defined testbench coverage collector *(non-virtual subclass)*

`pure virtual` methods must be defined in each subclass *(non-virtual subclasses)*

Almost exact same prototype required for `write` function

The class parameter `type` (`trans1`) must be used in this method prototype

# Pure Virtual Method Usage

- Pure virtual methods serve two important purposes:
  - Pure virtual methods can only be a method prototype

**Pure virtual methods create a place-holder**

  - Pure virtual methods must be overridden in a non-virtual class

**Imposes the *requirement* that it must be overridden in a non-virtual class**

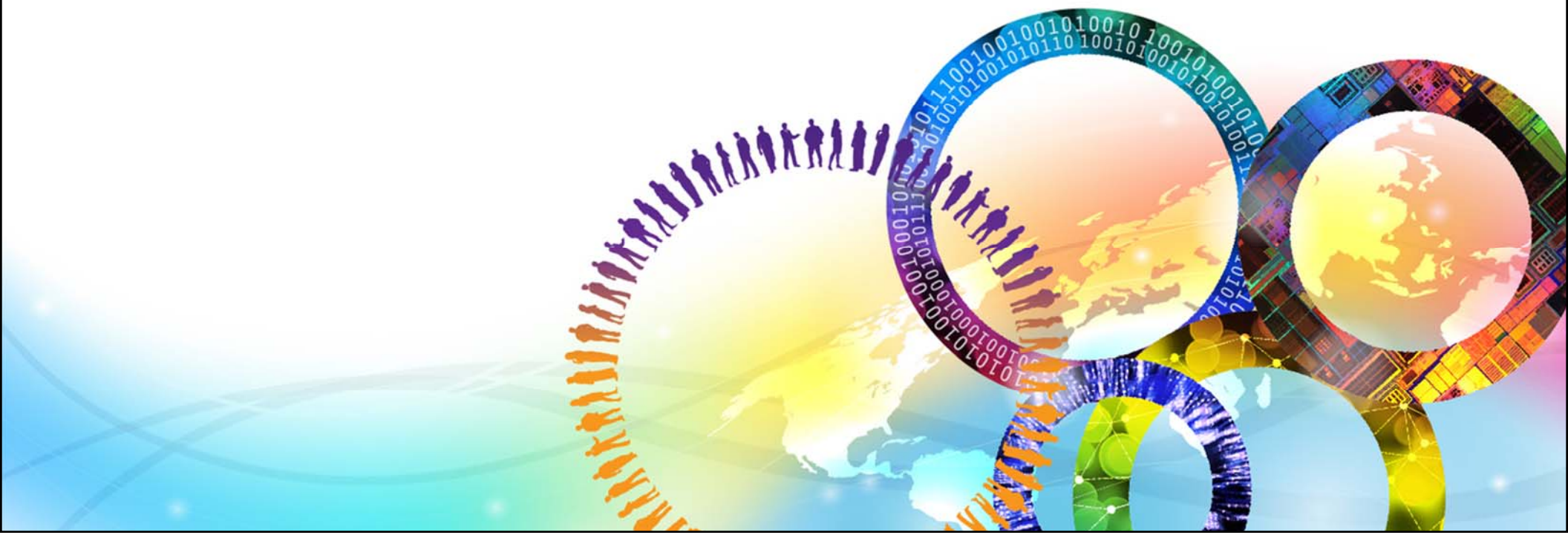
**Guideline #4:** **Declare a `pure` method whenever it is important to force a derivative class to implement the method, as is done by the `uvm_subscriber` virtual class**

Sunburst Design

snug
Synopsys Users Group
Silicon Valley

# Virtual Interfaces

## Techniques & Usage



# Virtual Interface Styles

- VIF styles:
  - Pass an interface handle down through class constructors for use by a test

**Somewhat straight-forward passing of `vif` handles through constructors, but tedious**

**Shown in the paper** *(not shown in this presentation)*

**Older OVM style**

  - Use a wrapper class and config object table to store and retrieve the wrapper

**The `vif` is stored in the wrapper**

**Wrapper class was required by OVM and uses upcasting and downcasting to store the `vif` handle**

**Recommended UVM-style**

  - Store the **`vif`** handle into the **`uvm_config_db`**. Use these methods:

```
uvm_config_db#(virtual dut_if)::set(null, "*", "vif" dif);
uvm_config_db#(virtual dut_if)::get(this, "",  "vif" vif);
```

**Typically stored from `top` module**

**Typically retrieved by driver and monitor**

**Simple technique added to UVM (not available in OVM)**

# Requirements for Virtual Interfaces

- 3 Requirements
  - Top module, DUT and real interface to communicate with the DUT

**An instantiated copy of the interface**

  - You will need a virtual interface to communicate with the test class

**A virtual interface handle declared in a class**

  - You need a way to tie the real interface to the virtual interface

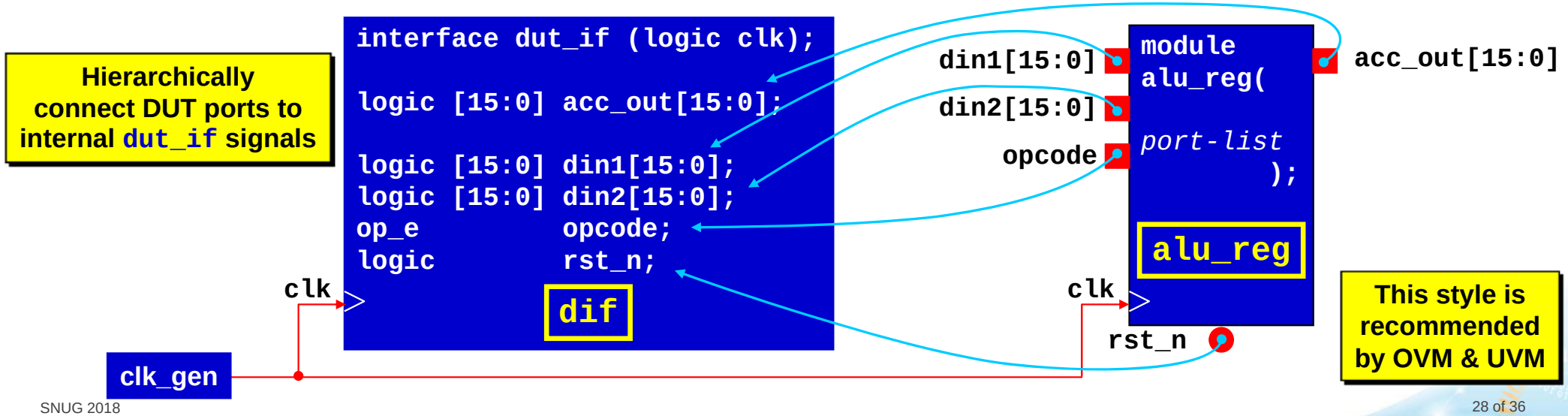
**You need a way to pass a real interface handle to a virtual interface handle in a class object**

# Interface connections to DUT ports

## alu_reg w/ Interface Example

```
module top;
  logic clk;
  clkgen  clkgen  (.*);
  dut_if  dif     (.*);
  alu_reg alu_reg (.acc_out(dif.acc_out), .din1(dif.din1),
                   .din2(dif.din2),   .opcode(dif.opcode),
                   .rst_n(dif.rst_n), .*);
  ...
endmodule
```

```
module alu_reg (
  output logic [15:0] acc_out,
  input  logic [15:0] din1, din2,
  input  op_e         opcode,
  input  logic        clk, rst_n);


  ...
endmodule
```



# OVM-Style - Wrapper Class Needed

```
module top;
  ...
  dut_if          dif (clk);

  dut_if_wrapper dif_w;
  ...
  initial begin
    dif_w = new(dif);
    set_config_object("*", "dif_w", dif_w, 0);
    run_test();
  end
endmodule
```

The static `dif` handle cannot be stored in a class-based table

A wrapper-class is required to store the static `dif` handle

Pass the `dif` handle to the wrapper constructor

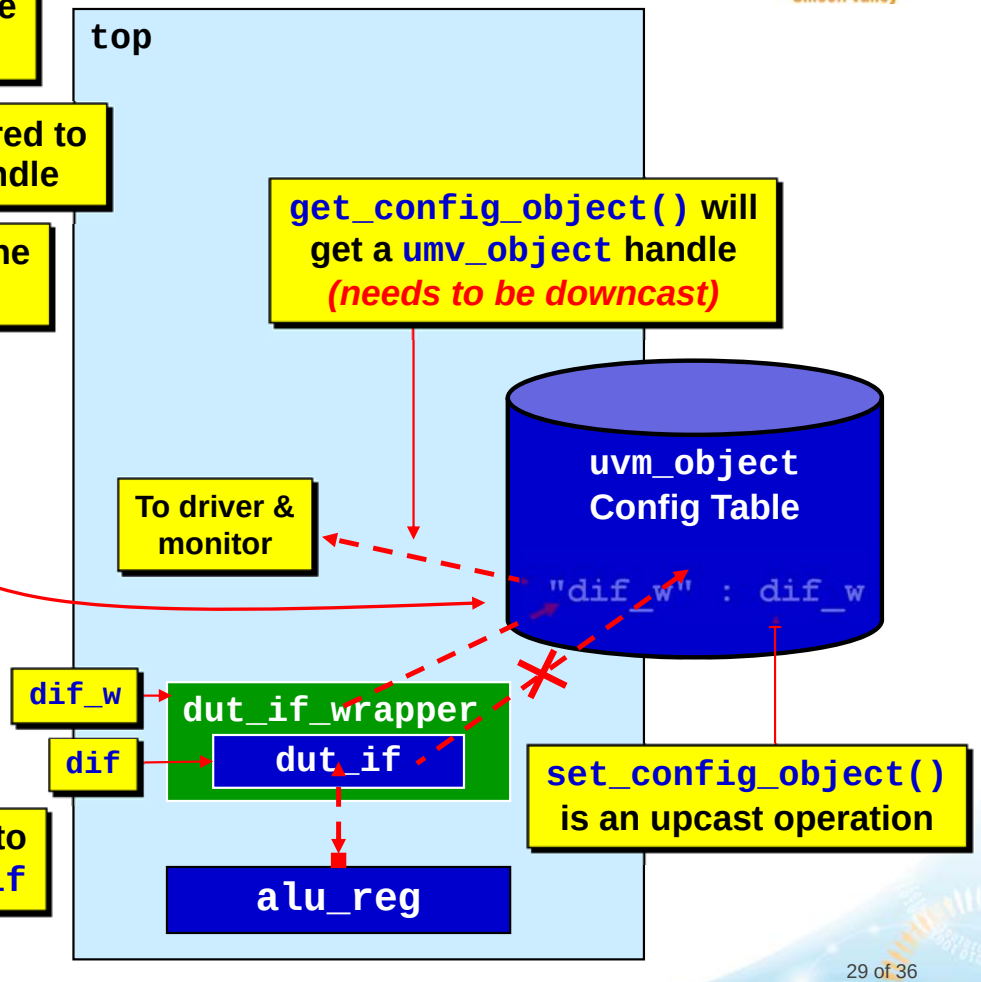
Store the `dif_w` handle into the `uvm_object` config table

```
class dut_if_wrapper extends uvm_object;
  virtual dut_if vif;

  function new (virtual dut_if nif);
    vif = nif;
  endfunction
endclass
```

`vif` handle storage

This `dut_if` handle is local to the function - it cannot be `vif`

`nif` must be copied to `vif`

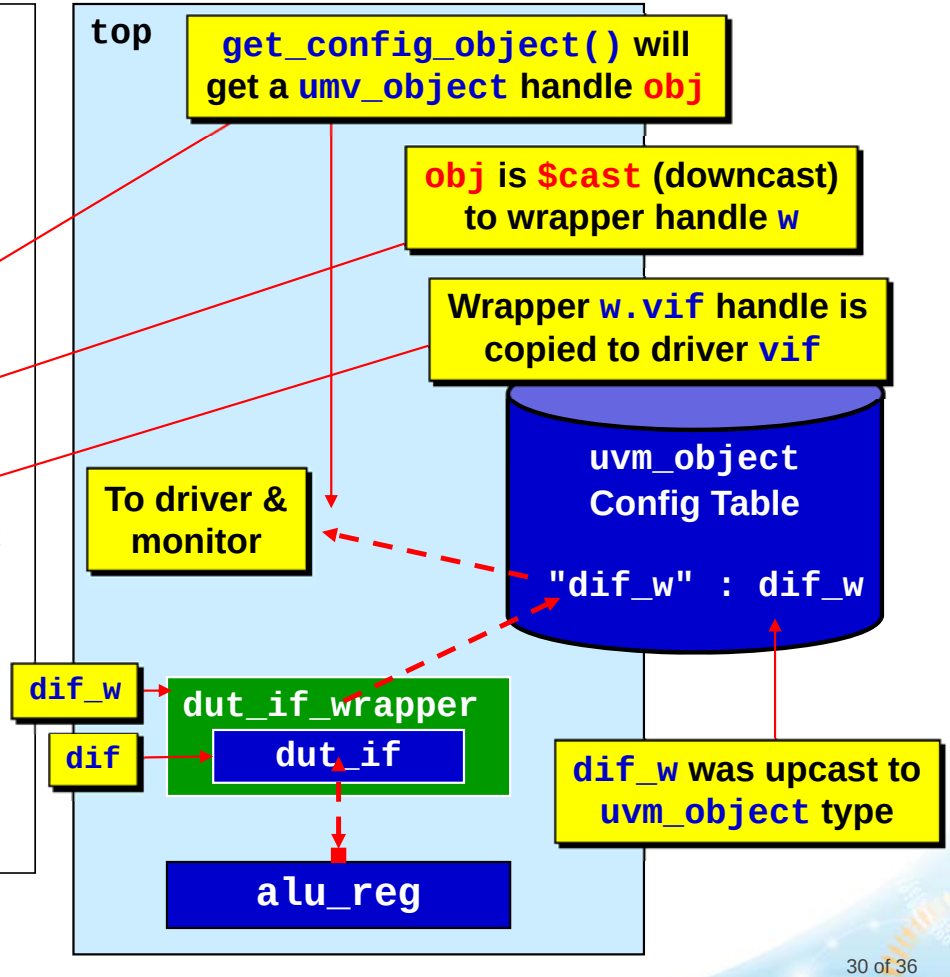


# OVM-Style - Driver Gets vif Handle

```
class tb_driver extends uvm_driver #(trans1);
  …
  virtual dut_if vif;
  …
  function void build_phase(uvm_phase phase);
    super.build_phase(phase);

    uvm_object     obj;
    dut_if_wrapper w;
    //------------------------------
    if (!get_config_object("dif_w", obj, 0))
      `uvm_fatal("NOVIF",{"vif must be set for:",
                          get_full_name(),".vif"});

    if (!$cast(w, obj))
      `uvm_fatal("NOVIF",{"bad dif_w handle for",
                          get_full_name()});

    vif = w.vif;
  endfunction
  …
endclass
```

Driver will use `vif` handle to drive signals to `dut_if`

Monitor uses the same technique to retrieve the `vif` handle



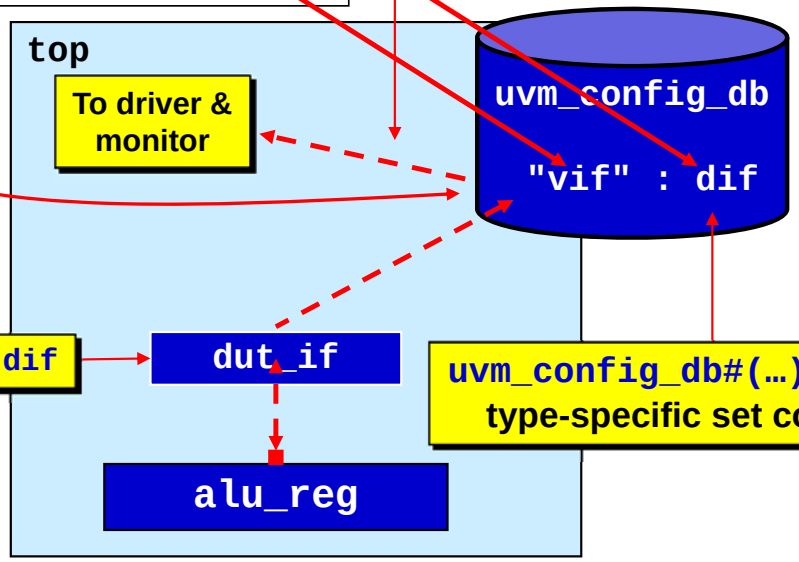
# UVM-Style - uvm_config_db

```
module top;
  ...
  dut_if          dif (clk);
  ...
  initial begin
    uvm_config_db#(virtual dut_if)::set(null, "*", "vif", dif);
    run_test();
  end
endmodule
```

The static dif handle to be stored in uvm_config_db

Store the dif handle into the uvm_config_db

uvm_config_db#(…)::get(…);




uvm_config_db#(…)::set/get(…) greatly simplifies storing & retrieving the dif handle

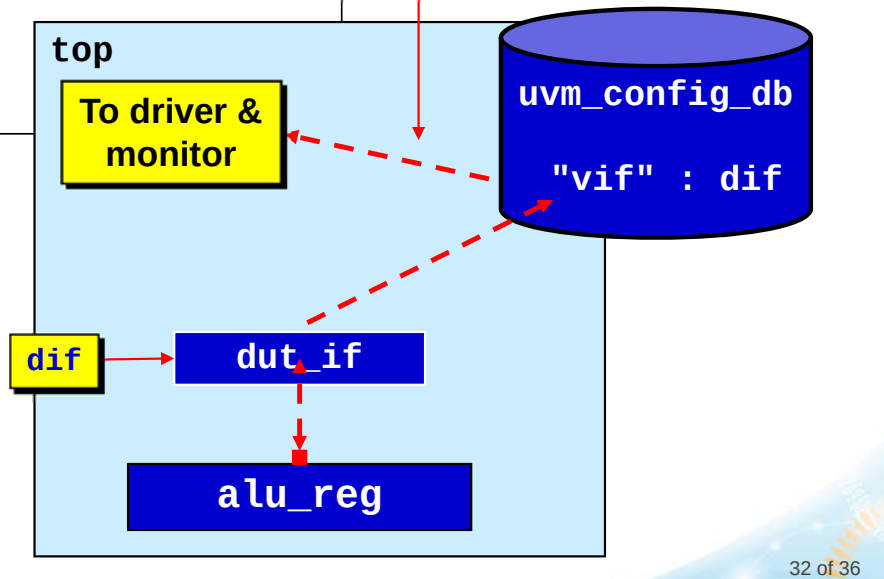
# UVM-Style - Driver Gets uvm_config_db

```
class tb_driver extends uvm_driver #(trans1);
  …
  virtual dut_if vif;
  …
  function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    if (!uvm_config_db#(virtual dut_if)::get(this, "", "vif", vif))
      `uvm_fatal("NOVIF", {"vif must be set for: ",
                           get_full_name(), ".vif"});
  endfunction
  …
endclass
```

Driver will use `vif` handle to drive signals to `dut_if`

Get the `dif` handle that was stored in the `uvm_config_db` and assign it to the local `vif` field

`uvm_config_db#(…)::get(…);`




Much easier than using the wrapper and downcasting!

Monitor uses the same technique to retrieve the `vif` handle

# UVM config_db Sumary

- The **`uvm_config_db`** commands eliminate the need to create a separate wrapper class to store the static interface handle

- The **`uvm_config_db`** also and eliminates the need to do upcasting and downcasting to save the **`vif`**

**This is the UVM preferred style to set and get the `vif` handle**

**Guideline #5:** Use **`uvm_config_db#(virtual dut_if)::set(…)`** and **`uvm_config_db#(virtual dut_if)::get(…)`** commands to store and retrieve an interface for use by a UVM testbench

# Summary of Guidelines

Guideline #1: Class methods should be **`function`** and **`void function`**.
Only use **`task`** when the method consumes simulation time

Guideline #2: Declare SystemVerilog class methods to be **`virtual`** methods
unless there is a very good reason to prohibit method-polymorphism

Guideline #3: Declare local transaction handles using distinct names such as **`tr`** and
avoid local transaction handle names such as **`rhs_`**

Guideline #4: Declare a **`pure`** method whenever it is important to force a derivative class
to implement the method, as is done by the **`uvm_subscriber`** virtual class

Guideline #5: Use **`uvm_config_db#(virtual dut_if)::set(…)`** and
**`uvm_config_db#(virtual dut_if)::get(…)`** commands
to store and retrieve an interface for use by a UVM testbench

# Acknowledgements

*Thanks!*


- John Dickol and Don Mills for their reviews and comments of the paper and slides

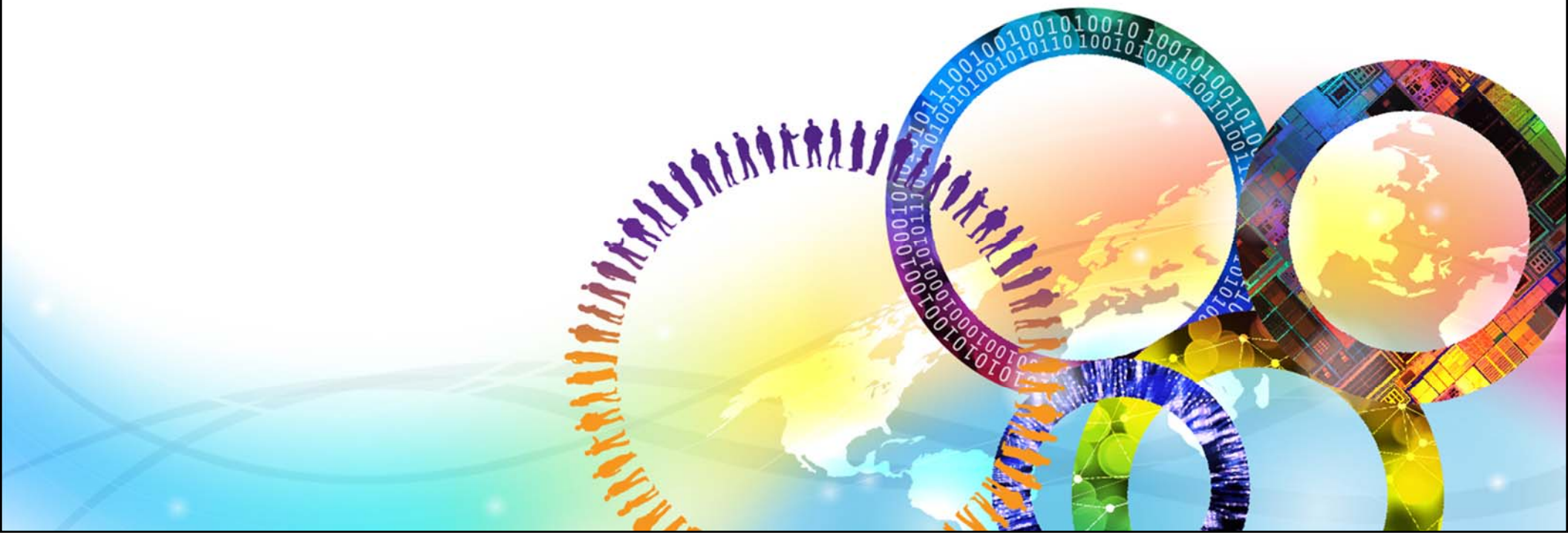
- Jeff Vance for discussions on advanced Virtual Interface techniques

**Their patience and efforts greatly improved the final paper and presentation!!**

Sunburst Design

snug
Synopsys Users Group
Silicon Valley

# Thank You



# SystemVerilog Virtual Classes, Methods, Interfaces

## Their Use in Verification and UVM


Clifford E. Cummings
Sunburst Design, Inc.

Heath Chambers
HMC Design Verification


**World-class SystemVerilog & UVM Verification Training**

**Life is too short for bad or boring training!**

March 21-22, 2018
Silicon Valley