

Object-Oriented Programming (OOP) in SystemVerilog

Part 2: Inheritance, Polymorphism & Reuse

Introduction

With the completion of earlier phases:

- **Day 1** explained *why* modern verification needs structure
- **Day 2** explained *how* SystemVerilog models data and execution
- **Day 3** explained *how behavior is abstracted and reused over time*
- **Day 4** explained *safe, race-free testbench–DUT interaction*
- **Day 5** introduced *object-oriented transaction modeling*

Day 6 completes the OOP foundation required for scalable verification.

At this stage, we move beyond *creating objects* to **reusing behavior safely and correctly at scale**. Modern verification environments **cannot survive without reuse**. Inheritance and polymorphism are not optional features - they are **core verification survival mechanisms**.

Why OOP Part 2 Is Critical in Verification

In real verification environments:

- You do not verify only one transaction type
- You do not write one driver per protocol variation
- You do not duplicate code for every corner case

Instead, verification requires:

- A **common base behavior**
- Specialized behavior where needed
- Runtime flexibility without code duplication

This is exactly what **inheritance and polymorphism provide**.

Scope of Day 6 (OOP – Part 2)

Day 6 focuses on **reuse, extensibility, and correctness** in object-oriented verification.

This phase covers:

- Inheritance
- Base and derived classes

- Virtual methods
- Polymorphism
- Safe object substitution
- Deep copy (conceptual → practical understanding)

This phase intentionally does **not** introduce UVM yet — but it explains **why UVM is designed the way it is**.

Inheritance: Reuse Without Duplication

What Is Inheritance?

Inheritance allows a class to **reuse and extend** another class.
In verification terms:

- A **base class** defines *common transaction behavior*
- A **derived class** modifies or extends that behavior

Why Inheritance Is Needed in Verification

Consider a protocol with:

- Read transactions
- Write transactions
- Burst transactions
- Error transactions

All of these share:

- Address
- Data
- Common formatting rules

But each has **different behavior**.

Without inheritance:

- Code is duplicated
- Bugs multiply
- Maintenance becomes impossible

With inheritance:

- Common behavior is written once
- Variations are handled cleanly

Simple Inheritance Example

```
class packet;  
  rand bit [7:0] addr;  
  rand bit [31:0] data;  
  
  function void display();  
    $display("BASE packet: addr=%h data=%h", addr, data);  
  endfunction  
endclass
```

Derived class:

```
class write_packet extends packet;  
  rand bit write_enable;  
  
  function void display();  
    $display("WRITE packet: addr=%h data=%h we=%b",  
      addr, data, write_enable);  
  endfunction  
endclass
```

Key Insight

- write_packet **inherits all fields of packet**
- Only the *difference* is added
- This models **protocol variations cleanly**

Base Handles and Derived Objects

One of the most important verification concepts.

```
packet p;  
p = new write_packet();
```

What this means:

- p is a **base-class handle**
- The object created is a **derived-class object**

This is **legal and intentional** in verification.

Why?

Because drivers, monitors, and scoreboards must work with:

- Generic transactions
- Without knowing the exact type

The Problem Without Virtual Methods

Consider this code:

```
packet p;  
p = new write_packet();  
p.display();
```

Which display() is called?

Without virtual:

- Base class method executes
- Derived behavior is ignored
- Verification intent is lost silently

This is one of the **most dangerous verification bugs**.

Virtual Methods: Enabling Correct Runtime Behavior

What Is a Virtual Method?

A virtual method ensures that:

- The **object type**, not the handle type,
- Determines which method executes

Correct Virtual Method Example

```
class packet;  
  rand bit [7:0] addr;  
  rand bit [31:0] data;  
  
  virtual function void display();  
  $display("BASE packet: addr=%h data=%h", addr, data);  
endfunction  
endclass
```

Derived class:

```
class write_packet extends packet;  
  rand bit write_enable;  
  
  function void display();  
  $display("WRITE packet: addr=%h data=%h we=%b",  
    addr, data, write_enable);  
endclass
```

```
endfunction
endclass
Usage:
packet p;
p = new write_packet();
p.display();
```

Output:

WRITE packet: addr=.. data=.. we=..

Key Rule : In verification, base-class methods that may be overridden must almost always be declared virtual.

This is why UVM uses virtual methods everywhere.

Polymorphism: One Interface, Many Behaviors

What Is Polymorphism?

Polymorphism allows:

- The same handle
- To represent different behaviors
- Resolved at runtime

In verification:

- One driver
- Many transaction types
- Zero code duplication

Verification-Oriented Example

```
packet pkt;
if (mode == WRITE)
    pkt = new write_packet();
else
    pkt = new packet();

pkt.display();
```

The driver:

- Does not care what exact type it receives
- Behaves correctly anyway

This is **polymorphic verification**.

Deep Copy vs Shallow Copy

Shallow Copy (Default Behavior)

`p2 = p1;`

- Only handle copied
- Both refer to same object
- Extremely dangerous

Used incorrectly, this causes:

- Scoreboard corruption
- Random failures
- Non-reproducible bugs

Deep Copy (Conceptual)

`p2 = new();`

`p2.addr = p1.addr;`

`p2.data = p1.data;`

This ensures:

- New object created
- Data duplicated
- Objects are independent
- Safe behavior

Why This Matters in Verification

Objects flow through:

- Drivers
- Monitors
- Scoreboards
- Coverage collectors

If deep copy is not handled correctly:

- Expected data is modified accidentally
- Failures appear randomly
- Debug becomes impossible

This is why **verification engineers must understand deep vs shallow copy.**

Syntax does not save you here — understanding does.

Conceptual Rule for Verification Engineers

- Handles are **references**
- Objects are **shared unless copied explicitly**
- Deep copy must be **intentional and controlled**

Key Takeaways — Day 6 (OOP – Part 2)

- Inheritance enables reuse without duplication
- Base handles can refer to derived objects
- Virtual methods are mandatory for correct behavior
- Polymorphism enables generic verification components
- Shallow copy is dangerous by default
- Deep copy ensures data integrity
- OOP behavior matters more than syntax

What Comes Next?

Day 7 – Threads & Interprocess Communication

The next phase introduces:

- fork...join
- Events
- Semaphores
- Mailboxes

With OOP now established, we can safely explore:

Multiple objects executing concurrently and communicating correctly

This is where verification environments become truly powerful.