Object-Oriented Programming (OOP) in SystemVerilog

Part 1 : Transaction-Level Modeling & Abstraction

Introduction:

With the completion of the 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 *how the testbench safely connects to the DUT and avoids races*

Day 5 marks the entry into object-oriented verification modeling.

At this stage, verification moves beyond signals, procedural blocks, and interfaces.
Modern verification environments are built using **objects**, not wires.
This phase introduces **Object-Oriented Programming (OOP)** in SystemVerilog — not from a software perspective, but from a **verification engineering perspective**.

Why OOP Is Critical in Verification

In real verification environments:

- Stimulus is not a single signal toggle
- Results are not checked signal-by-signal
- Behavior is not written as one large procedural block

Instead, verification operates on **transactions**:

- Packets
- Commands
- Requests
- Responses

These transactions must be:

- Grouped logically
- Passed between components
- Stored, copied, compared, and reused

Signals cannot model this behavior effectively. Objects can model. This is why verification is inherently object-oriented.

Where OOP Fits in a SystemVerilog Verification Environment

Up to Day 4, verification operated primarily at the **signal level**:

- The **DUT** operates on signals
- The **interface** groups signals and controls timing
- The **testbench** drives and samples signals safely

However, **none of these layers understand verification intent**.

Verification intent includes:

- What kind of operation is being performed
- What data is being transferred
- What response is expected
- How multiple operations relate to each other

This gap is filled by **Object-Oriented Programming**.

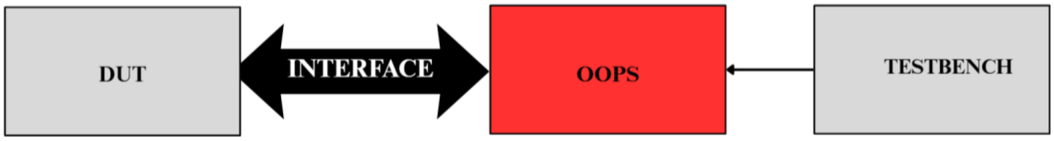

Fig. 1 Placement of OOPS layer in Verification Environment

This diagram illustrates a **fundamental architectural shift** in verification.

DUT (Design Under Test)

- Understands only signals
- Has no concept of packets, commands, or transactions
- Operates purely on hardware behavior

Interface

- Acts as a **signal-level communication boundary**
- Groups signals
- Enforces direction and timing
- Prevents race conditions

The interface solves *how* signals move — not *what* they represent.

OOP Layer (Transaction-Level Modeling)

The **OOP layer** is where verification becomes intelligent.

This layer:

- Models transactions instead of signals
- Encapsulates data and behavior together
- Separates *what* is being verified from *how* it is driven

Examples of what the OOP layer represents:

- A packet on a bus
- A read or write transaction
- A protocol command
- An expected response

These concepts **cannot be represented cleanly using signals or procedural code alone**.

Testbench Control

The testbench:

- Creates objects
- Calls methods
- Controls test scenarios
- Sequences verification intent
-

The testbench **does not toggle signals directly**. Instead, it operates on **objects**, which are then translated into signal activity through the interface.

Why OOP Sits Between Interface and Testbench

Conceptually:
The DUT never talks to the testbench directly. The testbench never talks to signals directly.

Instead:

- The testbench talks to **objects**
- Objects interact with the **interface**
- The interface drives the **DUT**

This separation provides:

- Clean abstraction
- Reusability
- Maintainability
- Scalability

This architecture is the foundation of all modern verification methodologies, including **UVM**.

Scope of Day 5 (OOP – Part 1)

Day 5 introduces **OOP fundamentals used in verification**, focusing on *what OOP is* and *why verification depends on it*.

This phase covers:

- Classes
- Objects and handles
- Constructors (new)
- Methods
- Static class members
- Object copying (deep vs shallow – introduction)

In this phase, **inheritance and polymorphism are intentionally NOT covered**. They will be introduced in DAY 6, **OOP – Part 2**.

Classes: The Core OOP Construct

In SystemVerilog verification, a **class** is a template (or blueprint) that describes:

- **What data exists** (properties)
- **What can be done with that data** (methods)

Unlike signals or variables, a class does **not represent hardware**.
It represents **verification intent**.

Why Classes Are Needed in Verification

Consider a real verification problem:
You are verifying a bus protocol.
A single operation involves:

- Address
- Data
- Read/Write type
- Valid/Ready information
- Expected response

Trying to manage this using:

- Individual signals
- Large procedural blocks

quickly becomes unmanageable.

Instead, we group all related information into **one object** → a **transaction**.

This is exactly what classes are designed for.

Simple Class Example (Transaction Modeling)

```
class packet;
  rand bit [7:0]  addr;
  rand bit [31:0] data;

  function void display();
    $display("addr=%h data=%h", addr, data);
  endfunction
endclass
```

Explanation of above code:

class packet;

- Defines a new **user-defined type**
- Think of this as creating a *new data type*, like int or logic, but more powerful

rand bit [7:0] addr;
rand bit [31:0] data;

- These are **properties (data members)**
- rand means these fields can be randomized later
- This models **transaction data**, not signals

function void display();
  $display("addr=%h data=%h", addr, data);
endfunction

- This is a **method**
- Methods define behavior related to the data
- The method operates on the **same object's data**

Key Insight

This packet class:

- Does **not** describe wires
- Does **not** describe hardware
- Describes **verification intent**

"Send a packet with this address and this data"

Objects and Handles

This is where **most beginners get confused**.

What Is an Object?

An **object** is an actual instance created from a class.

What Is a Handle?

A **handle** is a reference (pointer) to that object.

Example: packet p1;

At this point:

- No object exists
- p1 is just an empty handle

p1 = new();

Now:

- Memory is allocated
- An object of type packet is created
- p1 points to that object

Visualization

Think of it like this:

- **Class** → Blueprint of a house
- **Object** → Actual house built from the blueprint
- **Handle** → Address of the house

Handles Are Copied, Not Objects

This single rule explains **most verification bugs**.

Example

```
packet p1;
packet p2;

p1 = new();
p1.addr = 8'h10;
p1.data = 32'hAAAA_BBBB;

p2 = p1;   // Handle copy
```

What Actually Happens?

- No new object is created
- Data is NOT duplicated
- Both p1 and p2 point to the **same object**

Example:

```
p2.data = 32'h1234_5678;
p1.display();
```

Output:

addr=10 data=12345678

Even though you changed p2, p1 also changed.

Why This Is Dangerous

- Scoreboards store expected packets
- Drivers modify packets
- Monitors reuse packets

If two components unknowingly share the same object:

- Data gets corrupted
- Failures become random
- Bugs become non-reproducible

This is why **object copying must be handled carefully**.

Constructors (new) — Why They Exist

Unlike signals or variables, **classes do not exist automatically**.

Example

packet pkt;

- pkt exists as a handle
- No memory allocated
- Accessing pkt.addr will cause a runtime error

Correct Usage

packet pkt;
pkt = new();

Now:

- Memory is allocated
- All properties exist
- Methods can be called safely

Key Difference from Variables

Feature	Variables	Objects
Memory allocated automatically	Yes	No
Requires new()	No	Yes
Passed by value	Yes	No (handle)

Static Members (High-Level View)

Static members belong to the **class**, not to individual objects.

Example

```
class packet;
  static int packet_count;
  bit [7:0] addr;

  function new();
```

```
    packet_count++;
  endfunction
endclass
```

Explanation

- packet_count is shared by **all packet objects**
- Every time new() is called, the same counter is incremented

Usage Example

```
packet p1 = new();
packet p2 = new();

$display("count=%0d", packet::packet_count);
```

Output:

```
count=2
```

Why Static Members Are Useful

- Global counters
- Statistics
- Configuration shared across components
- Debug tracking

Deep Copy vs Shallow Copy (Why This Matters)

Shallow Copy (Default Behavior)

```
p2 = p1;
```

- Only handle copied
- Both refer to same object
- Dangerous

Deep Copy (Conceptual)

```
p2 = new();
p2.addr = p1.addr;
p2.data = p1.data;
```

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

Full deep-copy implementations will be covered in **OOP – Part 2**

Why Verification Engineers Must Understand This

In verification:

- Objects flow through drivers, monitors, scoreboards
- Data integrity is critical
- Silent corruption is deadly

Understanding **object behavior** is more important than knowing syntax.

Key Takeaways (Day 5 – OOP Core)

- Classes model **verification intent**, not hardware
- Objects exist only after calling new
- Handles are references, not data
- Handle copies cause shared data bugs
- Static members are shared across objects
- Deep vs shallow copy determines correctness

What Comes Next?

📌 **Day 6 – OOP (Part 2): Inheritance, Polymorphism & Reuse**

The next phase will explain:

- Inheritance and base classes
- Virtual methods
- Polymorphism
- Why UVM relies heavily on these concepts