🧩 UVM Cheat Sheet

A quick reference guide covering core UVM concepts - what they are, why we use them, and how they fit into a verification testbench.

🧠 1. What is UVM? Why Do We Use It?

Definition:

UVM (Universal Verification Methodology) is a standardized, SystemVerilog-based methodology for building reusable, modular, and scalable verification environments.

Why We Use It:

- ✅ **Reusability:** Same components (driver, monitor, agent) can be reused across multiple projects.
- ✅ **Scalability:** Works for block-level to SoC-level verification.
- ✅ **Automation:** Handles complex tasks like sequences, coverage, and reporting.
- ✅ **Standardization:** Common structure across teams → easier debugging and collaboration.

In short:

> UVM gives you a professional, organized way to test chips instead of writing ad-hoc testbenches from scratch.

🧱 2. Difference: SystemVerilog vs. UVM Testbench

Aspect	SystemVerilog Testbench	UVM Testbench
Structure	Flat, ad-hoc	Hierarchical (test → env → agent → drv/seqr/mon)

Reuse	Manual (copy/paste)	Built-in via UVM factory & components
Stimulus	Procedural (`initial` blocks)	Class-based (sequences on sequencers)
Checking	Manual `$display` or assertions	Monitors + Scoreboards + Coverage
Flow control	Manual delays	UVM phases (build, connect, run, etc.)
Reporting	`$display`	UVM report macros (info, warning, error, fatal)

✅ **Reason:** UVM provides structure, modularity, and reusability — essential for complex designs.

🧩 3. Testbench Architecture (UVM Layers)

```
UVM Testbench
└── uvm_test
    └── uvm_env
        └── uvm_agent
            ├── uvm_driver
            ├── uvm_sequencer
            └── uvm_monitor
```

Component	**Role**
test	Top-level, controls configuration & scenario
env	Holds multiple agents, scoreboard, coverage
agent	Groups driver, monitor, sequencer for one interface
driver	Drives signals to DUT via virtual interface
sequencer	Sends transaction-level stimulus to driver
monitor	Observes DUT signals and sends data t scoreboard
scoreboard	Checks DUT outputs against expected behavior

🧠 4. UVM Class Hierarchy

Base Class	Used For	Example Classes	Phases ?
`uvm_component`	Structural hierarchy	env, agent, driver, monitor, test	✅ Yes
`uvm_object`	Lightweight data objects	transaction, sequence item	❌ No

✅ **Reason:** Components form the testbench structure; objects pass data between them.

🎬 5. Simulation Phases Overview

Phase	Purpose	Example Actions
build_phase	Create components & configure objects	`drv = my_driver::create()`
connect_phase	Connect TLM ports, exports	`driver.seq_item_port.connect(seqr.seq_item_export)`
run_phase	Time-based phase where simulation runs	`seq.start(seqr)`
extract_phase	Collect results from monitors/scoreboard	Count transactions, coverage
check_phase	Verify results (pass/fail)	Check scoreboard errors
report_phase	Print summary report	`UVM_INFO` summary

✅ **Reason:** Phases provide consistent simulation flow across all components.

⚙️ 6. Configuration Database (`uvm_config_db`)

Purpose:

A global database to share configuration handles (like virtual interfaces) between components without hard-coded paths.

Example:

```
// In test
i = my_if_if;
uvm_config_db#(virtual my_if)::set(null, "uvm_test_top.env.agent.*", "vif", i);

// In driver
uvm_config_db#(virtual my_if)::get(this, "", "vif", vif);
```

✅ **Reason:** Makes environments reusable and avoids hard-coding signal connections.

🏭 7. UVM Factory & Overrides

Factory: Creates objects/components dynamically using:

```
obj = my_class::type_id::create("obj", this);
```

Type Override (global)

```
factory.set_type_override_by_type(base::get_type(), derived::get_type());
```

➡️ Replaces all instances of `base` with `derived`.

Instance Override (specific path)

```
factory.set_inst_override_by_name("base", "derived", "uvm_test_top.env.agent.drv");
```

➡️ Replaces only the specific instance.

✅ **Reason:** Enables flexible swapping of components without editing source code → reusability.

🧩 8. UVM Macros Overview

Macro	Used For	Explanation	Example
`uvm_object_utils(TYPE)`	Register non-component objects	Enables factory creation for transactions	`pkt::type_id::create("tr")`
`uvm_component_utils(TYPE)`	Register components	Enables factory creation in hierarchy	`my_driver::type_id::create("drv", this)`
`uvm_do(tr)`	Sequence shorthand	`create + start + randomize + finish`	\uvm_do_with(tr, {addr==8'hA5;})`

✅ **Reason:** Macros save typing, register classes for factory use, and ensure consistent UVM structure.

🧾 9. UVM Reporting Macros

Macro	Purpose	Stops Sim?	Example
`uvm_info(ID, MSG, VERB)`	Info/debug messages	❌	`UVM_INFO("DRV", "Driving data", UVM_MEDIUM)`
`uvm_warning(ID, MSG)`	Warning	❌	`UVM_WARNING("TIMING", "Late signal")`
`uvm_error(ID, MSG)`	Error (counts)	❌	`UVM_ERROR("PROT", "Protocol violation")`
`uvm_fatal(ID, MSG)`	Fatal stop	✅	`UVM_FATAL("CFG", "Missing interface")`

Controlling Verbosity

Command-line: `+UVM_VERBOSITY=UVM_HIGH`
Runtime: `uvm_top.set_report_verbosity_level(UVM_HIGH);`

✅ **Reason:** Gives structured, filterable logs instead of raw `$display` — easier debugging.

🎬 10. UVM Objections

Purpose:

Controls **when simulation ends** automatically.

```
task run_phase(uvm_phase phase);
  phase.raise_objection(this);
  seq.start(seqr);
  phase.drop_objection(this);
endtask
```

- `raise_objection(this)` → keeps simulation running.
- `drop_objection(this)` → allows simulation to end.
- UVM ends sim only when **all objections are dropped.**

Analogy:

Everyone raises a hand ✋ when busy — simulation ends only when all hands go down.

✅ **Reason:** Ensures all components finish their work before simulation stops.

🧘 Recap Table

Concept	Description	Reason
UVM	Reusable verification framework	Standardized & scalable testing
SV vs. UVM	SV = manual, UVM = structured	UVM simplifies reuse & debugging
Testbench Arch	test→env→agent→drv/mon/seqr	Modular and hierarchical
Hierarchy Classes	`uvm_component`, `uvm_object`	Components = structure, Objects = data
Phases	build→connect→run→report	Defines simulation flow
Config DB	Shared settings via DB	Avoid hard-coded connections
Factory & Overrides	Dynamic creation & replacement	Reusability and flexibility
Macros	Boilerplate shortcuts	Less typing, consistent factory use
Reporting Macros	Structured logging	Easier debug with verbosity control
Objections	Simulation control	Coordinated ending for all components

---

🌟 Final Takeaway

UVM brings **structure, reusability, and control** to complex verification.
Learn each concept step-by-step, and soon they'll all connect naturally into one complete flow!