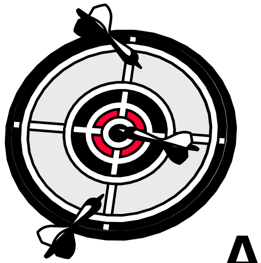
Agenda

DAY
3

© 2016 Synopsys, Inc. All Rights Reserved

Unit Objectives



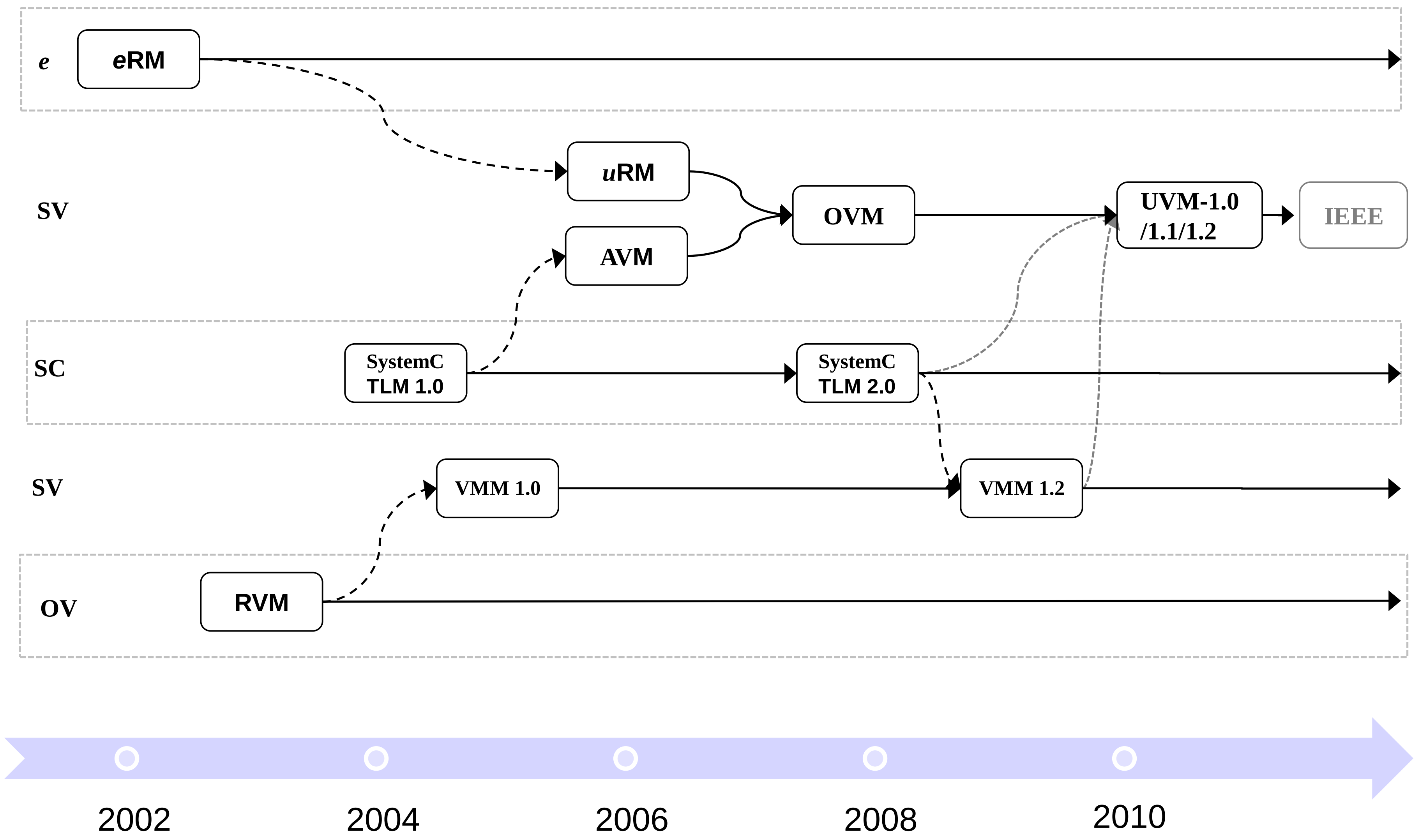
After completing this unit, you should be able to:

- Describe the UVM testbench architecture
- Describe the UVM environment execution sequence (phasing)

UVM - Universal Verification Methodology

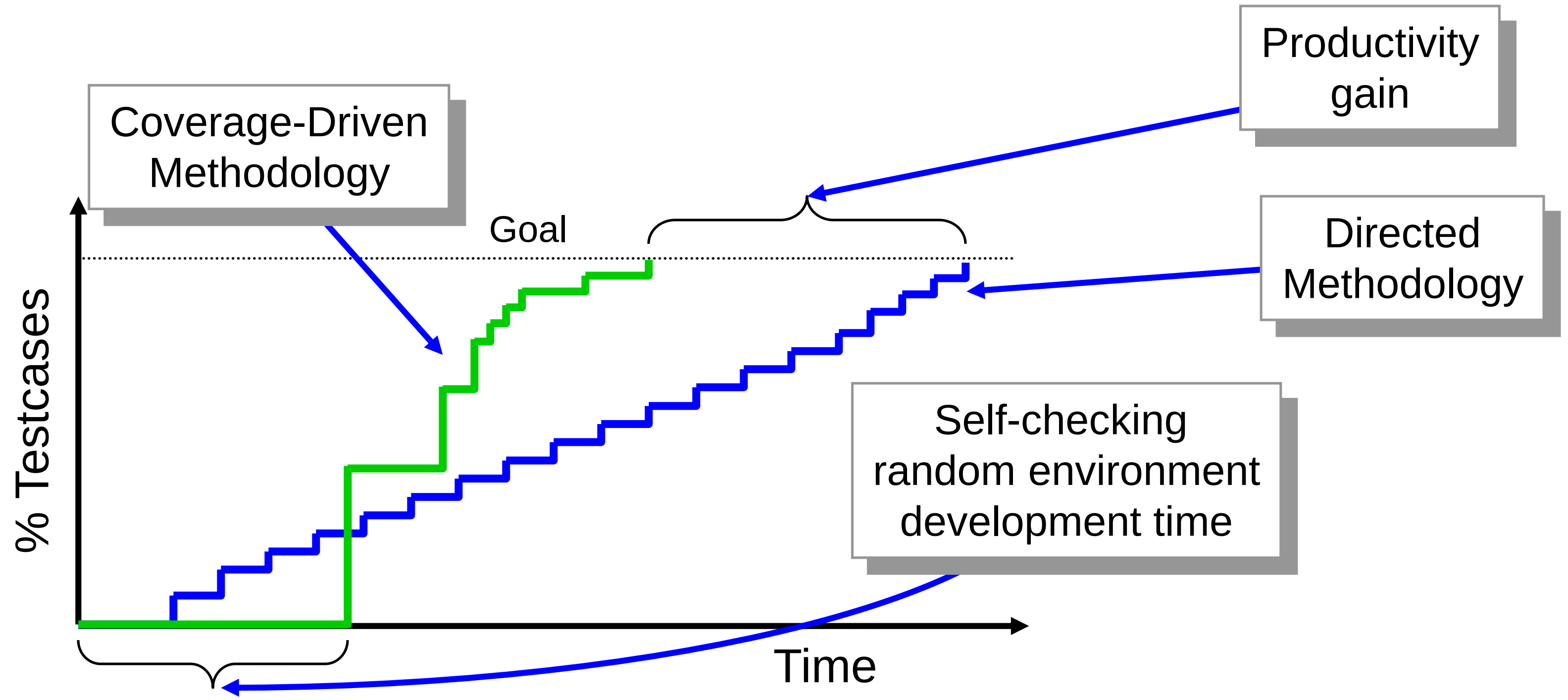
- **An effort (by an Accellera committee) to define a standard verification methodology & base class library**
  - Uses classes and concepts from VMM, OVM
  - Still work in progress
  - Published after all vendors' approval
- **Related Websites:**
  - Public Website - http://www.accellera.org/activities/vip/
  - Member Website - http://www.accellera.org/apps/org/workgroup/vip/
  - Mantis (Bug Tracking) - http://eda.org/svdb/view_all_bug_page.php
  - Sourceforge - http://uvm.git.sourceforge.net
  - UVM World
    - http://www.uvmworld.org/
    - http://www.uvmworld.org/forums/

Origin of UVM



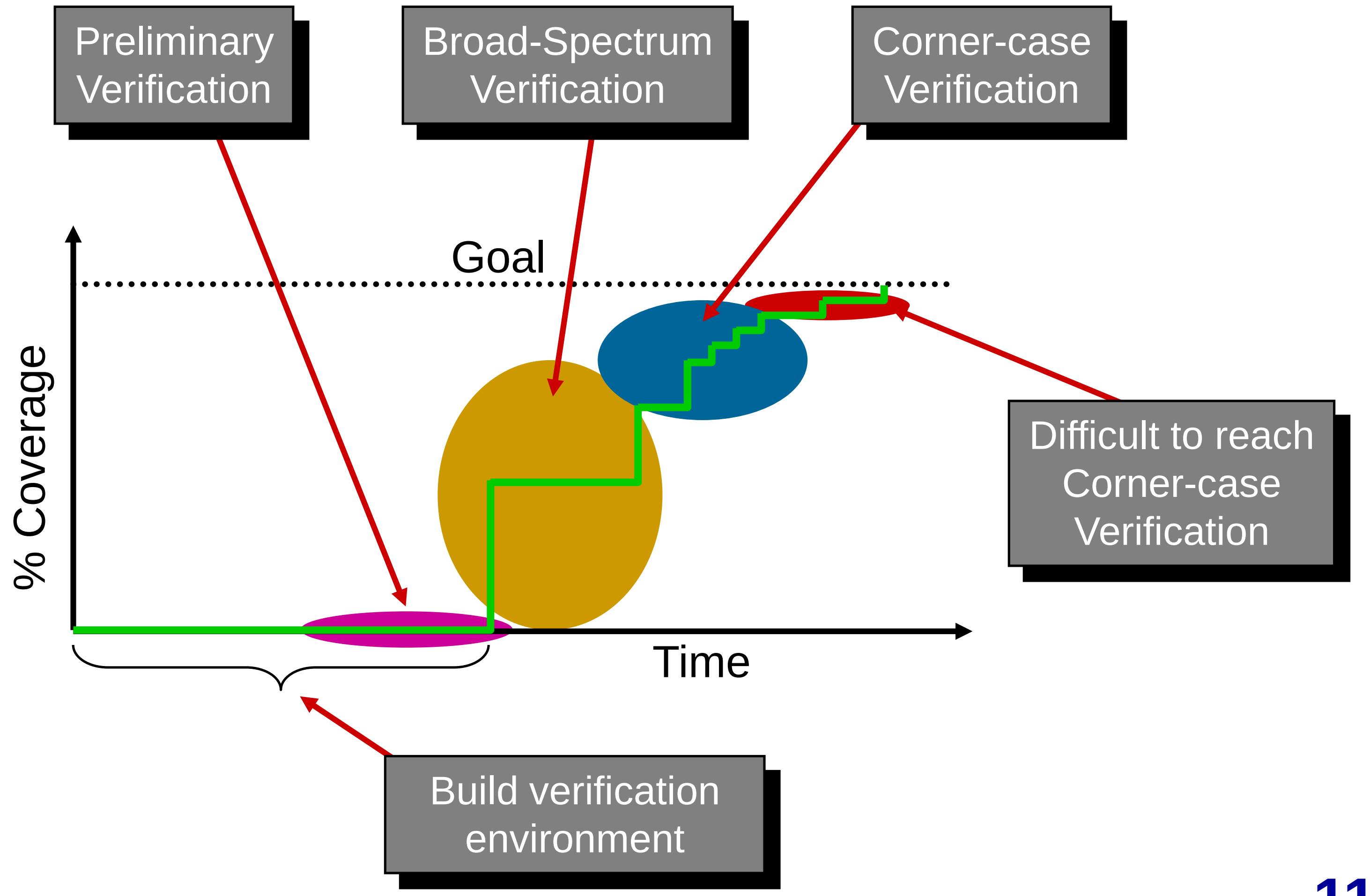
Coverage-Driven Verification

- Focus on uncovered areas
- Trade-off authoring time for run-time
- Progress measured using functional coverage metrics



Phases of Verification

Start with fully random environment. Continue with more and more focused guided tests

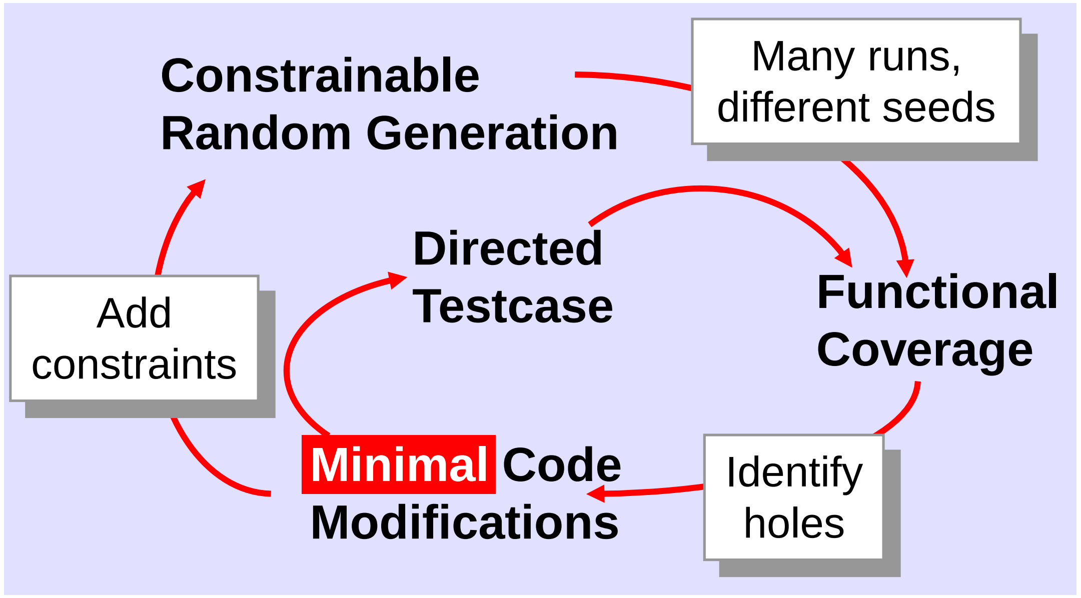

Preliminary Verification
Broad-Spectrum Verification
Corner-case Verification
Goal
% Coverage
Time
Difficult to reach Corner-case Verification
Build verification environment

Run More Tests, Write Less Code

- **Environment and component classes rarely change**
  - Sends good transactions as fast as possible
  - Keeps existing tests from breaking
  - Leave "hooks" so test can inject new behavior
    - Virtual methods, factories, callbacks
- **Test extends testbench classes**
  - Add constraints to reach corner cases
  - Override existing classes for new functionality
  - Inject errors, delays with callbacks
- **Run each test with hundreds of seeds**

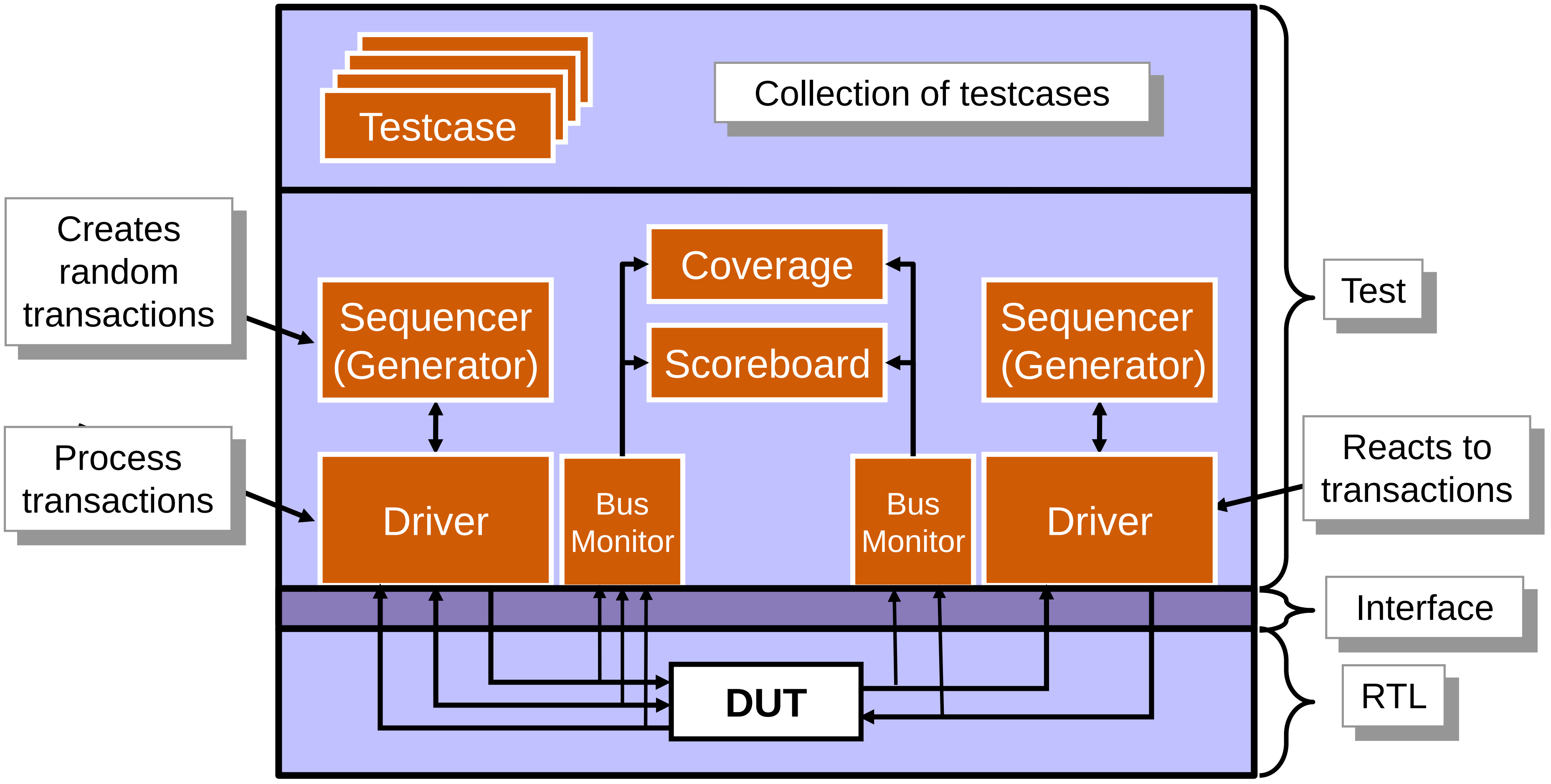
UVM Guiding Principles

- **Top-down implementation methodology**
  - Emphasizes "*Coverage Driven Verification*"
- **Maximize design quality**
  - More testcases
  - More checks
  - Less code
- **Approaches**
  - Reuse
    - Across tests
    - Across blocks
    - Across systems
    - Across projects
  - One verification environment, many tests
  - Minimize test-specific code


Constrainable Random Generation
Many runs, different seeds
Directed Testcase
Functional Coverage
Add constraints
Minimal Code Modifications
Identify holes

The Testbench Environment/Architecture

- **SystemVerilog testbench structure**



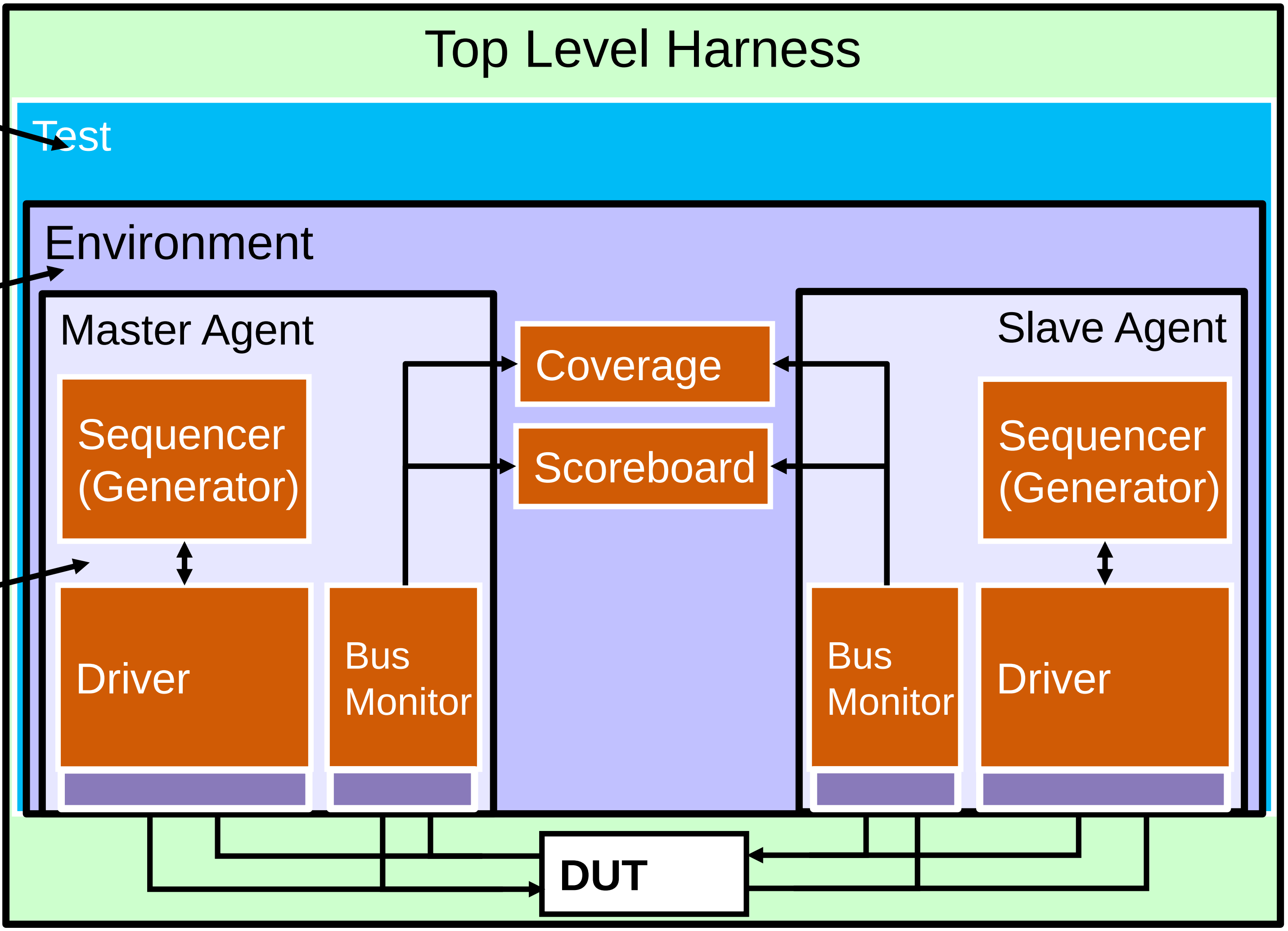
UVM Encourages Encapsulation for Reuse

- **Structure should be architected for reuse**

Test instantiates the environment and modifies the environment on a testcase by testcase basis

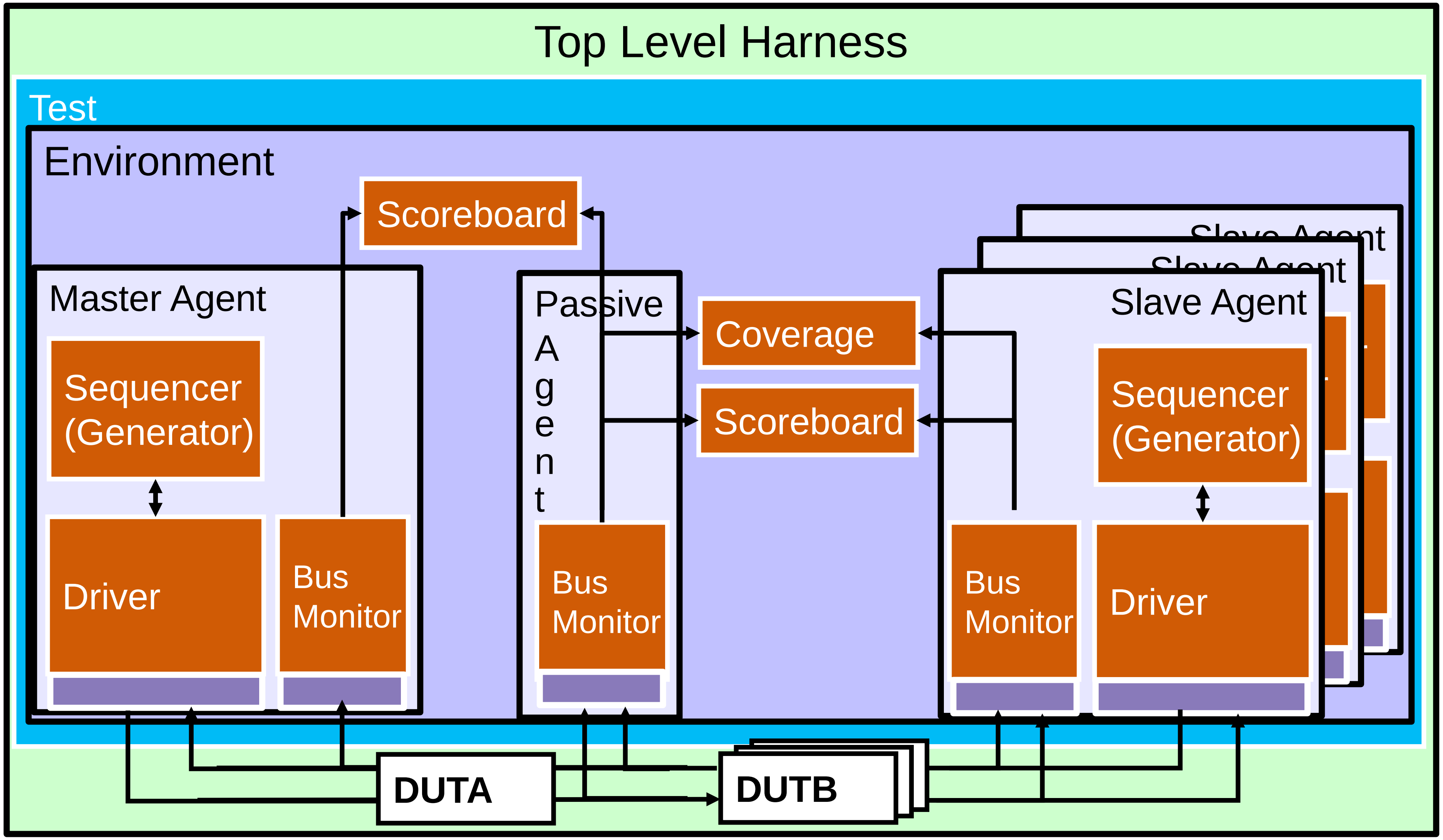
Agents, coverage and scoreboard should be encapsulated in an environment

Sequencer, driver and monitor associated with an interface should be encapsulated as an agent for that interface



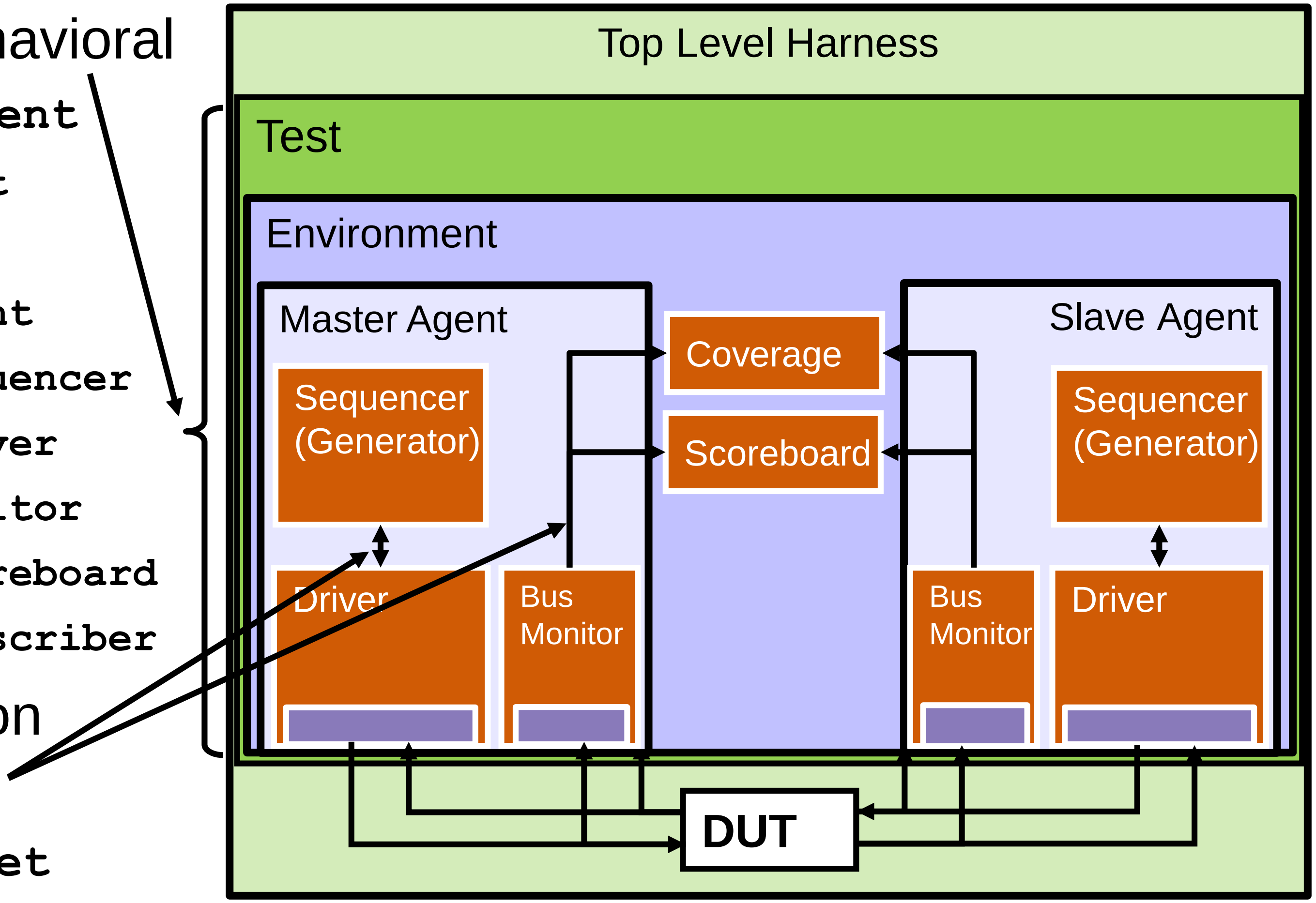
UVM Structure is Scalable

- **Agents are the building blocks across test/projects**



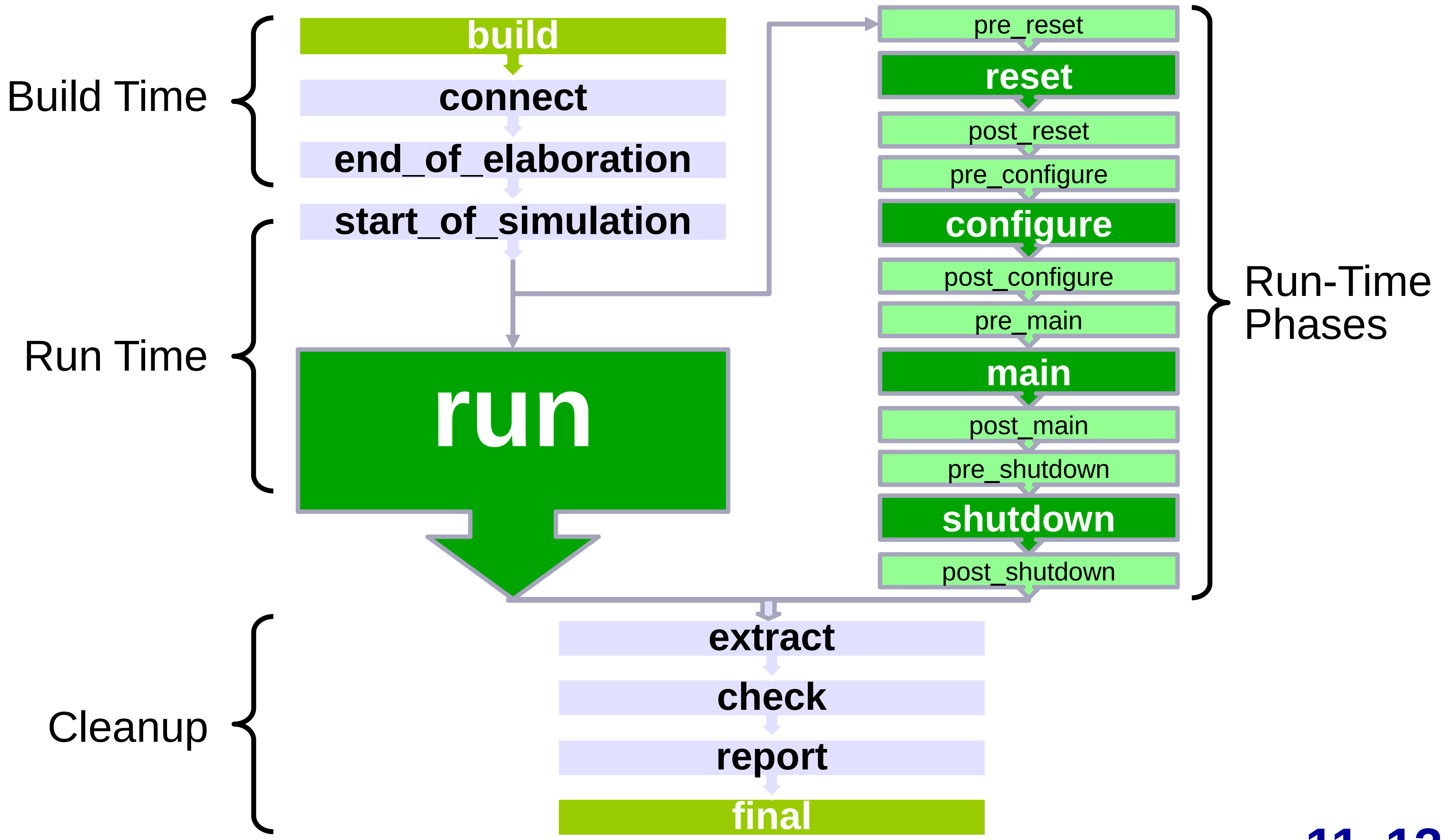
Standards: Structural Support in UVM

- **Base Classes provided by UVM**
  - Structural/Behavioral
    - `uvm_component`
      - `uvm_test`
      - `uvm_env`
      - `uvm_agent`
      - `uvm_sequencer`
      - `uvm_driver`
      - `uvm_monitor`
      - `uvm_scoreboard`
      - `uvm_subscriber`
  - Communication
    - `uvm_*_port`
    - `uvm_*_socket`
  - Data
    - `uvm_sequence_item`



Standards: Component Phasing

- UVM defines standard phases for **component synchronization** and **automatic execution**



Standards: Component Configuration

- **Component configuration using a resource database**

```
class router_env extends uvm_env;
  // utils macro and constructor not shown
  virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    uvm_config_db #(int)::set(this, "*.drv", "port_id", 10);
  endfunction
```

Set **`port_id`** value in resource database

```
class driver extends uvm_driver #(packet);
  // constructor not shown
  int port_id = -1;  // user configurable
  `uvm_component_utils_begin(driver)
    `uvm_field_int(port_id, UVM_DEFAULT)
  `uvm_component_utils_end

  virtual task build_phase(…); ...
    if (!uvm_config_db #(int)::get(this, "", "port_id", port_id))
      `uvm_info("DRVCFG", {get_full_name(), " using default port_id"},
                                                            UVM_MEDIUM);
    ...
```

Retrieve value from resource database

For debugging, print full name

Standards: Reporting and Handshaking

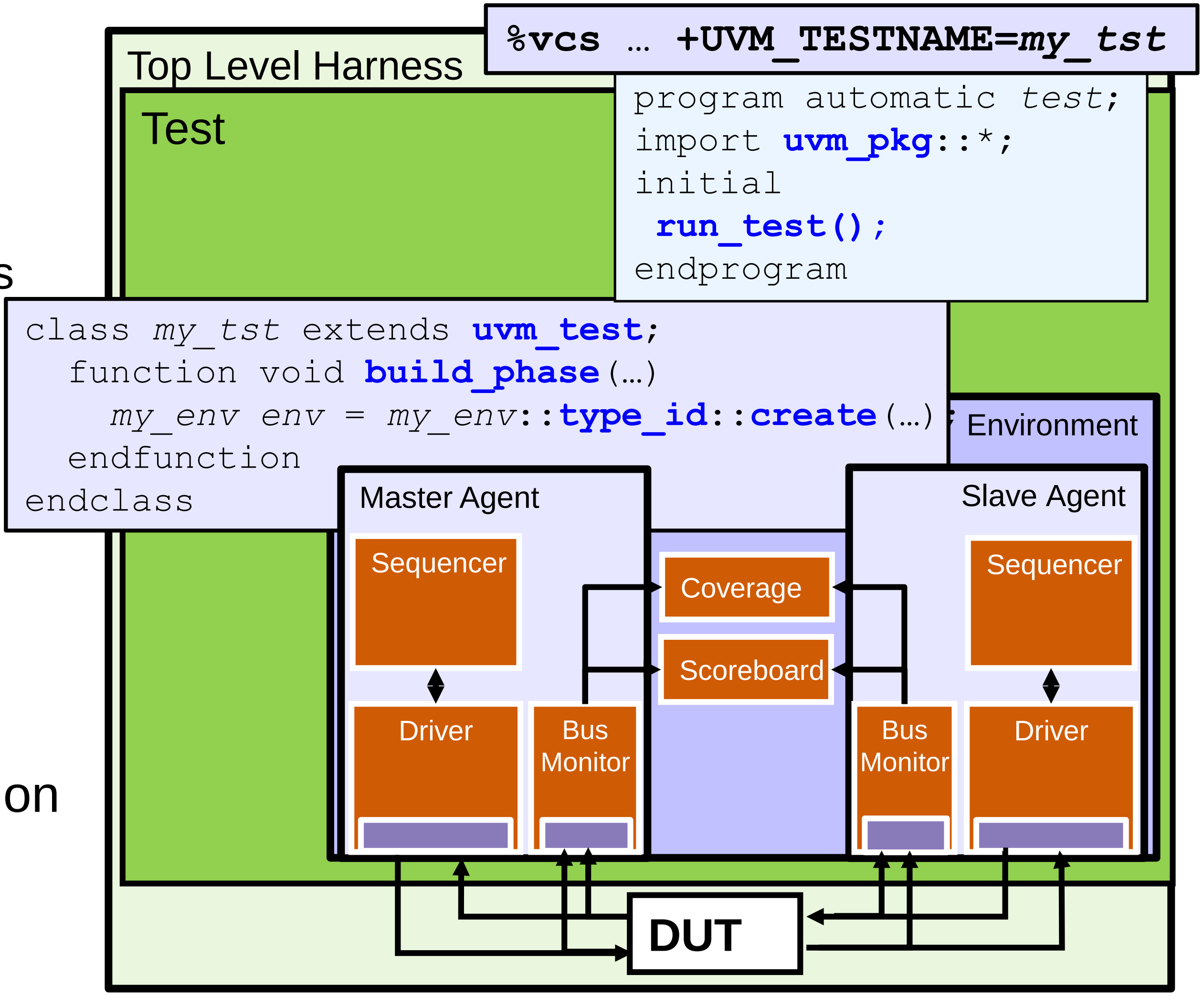
- **Standard message macros**
  - Can filter, promote or demote messages as needed on a global or instance basis

```
`uvm_fatal(string ID, string MSG);
`uvm_error(string ID, string MSG);
`uvm_warning(string ID, string MSG);
`uvm_info(string ID, string MSG, verbosity);
```

- **Standard handshaking mechanisms**
  - `uvm_event` class
    - Wait for one trigger to releases all waits
  - `uvm_barrier` class
    - wait for n waiters before opening barrier
  - `uvm_pool` class
    - common pool of user-defined resources for global lookup

Standards: Implementing UVM Test

- **Test encapsulates verification environment**
- **Instantiates**
  - Agents
    - Sequencers
    - Drivers/Monitors
  - Scoreboards
  - Coverage model
  - Signal Interfaces
- **Controls**
  - Configuration
  - Start of simulation
  - Phases of simulation
  - Pass/Fail report
  - Factory
- **Executed via `run_test()`**

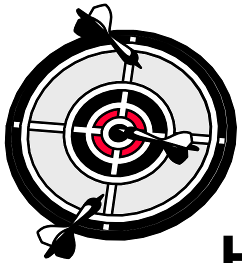


```
%vcs … +UVM_TESTNAME=my_tst
```

```
program automatic test;
import uvm_pkg::*;
initial
 run_test();
endprogram
```

```
class my_tst extends uvm_test;
  function void build_phase(…)
    my_env env = my_env::type_id::create(…);
  endfunction
endclass
```

Unit Objectives



Having completed this unit, you should be able to:

- Describe the UVM testbench architecture
- Describe the UVM environment execution sequence (phasing)

That's all Folks!