Chapter 8
Advanced OOP and Testbench Guidelines

How would you create a complex class for a bus transaction that also performs error injection and has random delays? The first approach is to put everything in a large, flat class. This approach is simple to build, easy to understand (all the code is right there in one class) but can be slow to develop and debug. Additionally, such a large class is a maintenance burden, as anyone who wants to make a new transaction behavior has to edit the same file. Just as you would never create a complex RTL design using just one Verilog module, you should break classes down into smaller, reusable blocks.

Another approach is composition. As you learned in Chapter 5, you can instantiate one class inside another, just as you instantiate modules inside another, building up a hierarchical testbench. You write and debug your classes from the top down or bottom up, always looking for natural partitions when deciding what variables and method go into the various classes. A pixel could be partitioned into its color and coordinate. A packet might be divided into header and payload. You might break an instruction into opcode and operands. See Section 8.4 for guidelines on partitioning.

Sometimes it is difficult to divide the functionality into separate parts. Consider injecting errors during a bus transaction. When you write the original class for the transaction, you may not think of all the possible error cases. Ideally, you would like to make a class for a good transaction, and later add different error injectors. The transaction has data fields and an error-checking checksum field generated from the data. One form of error injection is corruption of the checksum field. If you use composition, you need separate classes for good transactions and error transactions. Testbench code that used good objects would have to be rewritten to process the new error objects. What you need is a class that resembles the original class but adds a few new variables and methods. This result is accomplished through inheritance.

C. Spear and G. Tumbush, *SystemVerilog for Verification: A Guide to Learning the Testbench Language Features*, DOI 10.1007/978-1-4614-0715-7_8,

Inheritance allows a new class to be extended from an existing one by adding new variables and methods. The original class is known as the base class. Since the new class extends the capability of the base class, it is called the extended class. Inheritance provides reusability by overlaying features, such as error injection, on an existing class, without modifying that class.

A real power of OOP is that it gives you the ability to take an existing class, such as a transaction, and selectively update parts of its behavior by replacing methods, but without having to change the surrounding infrastructure. All your original tests that depend on the base class keep working, and you can now create new tests with the extended class. With some planning, you can create a testbench solid enough to send basic transactions, but able to accommodate any extensions needed by the test.

Note that this chapter goes into a wide range of advanced OOP topics, many of which you won't need when learning SystemVerilog. Feel free to skip the later sections for now, and save them for when you are digging into the internals of UVM and VMM.

8.1 Introduction to Inheritance

Figure 8.1 shows a simple testbench. The test controls the generator. The generator creates transactions, randomizes them, and sends them to the driver along the dotted line. The driver breaks down the transaction into pin wiggles and sends it into the DUT along the dashed line. The rest of the testbench is left out.

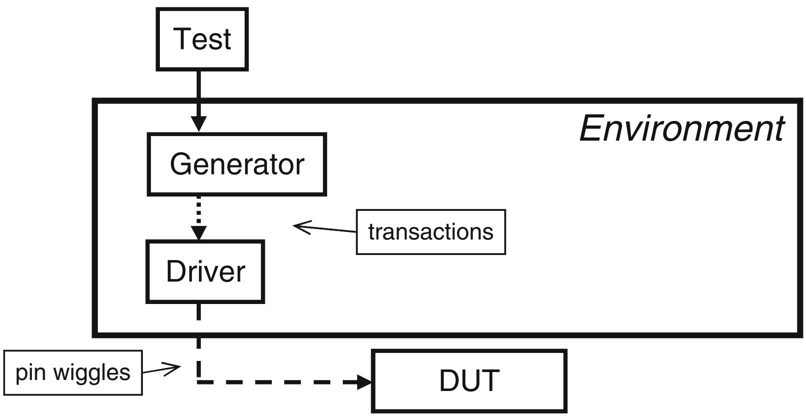

Fig. 8.1 Simplified layered testbench

8.1.1 Basic Transaction

The basic transaction class in Sample 8.1 has variables for the source and destination addresses, eight data words, and a checksum for error checking, plus methods for displaying the contents and calculating the checksum. The `calc_csm` function is tagged as `virtual` so that it can be redefined if needed, as shown in the next section. Virtual methods are explained in more detail later in this chapter in Section 8.3.2. The class is simple enough that it uses the default SystemVerilog constructor that allocates memory and initializes variables to their default value.

Sample 8.1 Base `Transaction` class

```
class Transaction;
  rand bit [31:0] src, dst, data[8];  // Random variables
  bit [31:0] csm;                     // Calculated variable

  virtual function void calc_csm();
    csm = src ^ dst ^ data.xor;
  endfunction

  virtual function void display(input string prefix="");
    $display("%sTr: src=%h, dst=%h, csm=%h, data=%p",
             prefix, src, dst, csm, data);
  endfunction
endclass
```

Normally calculating the checksum would be done in post_randomize(), but in this example it has been separated from the randomization to show how to inject errors.

Figure 8.2 shows a diagram for the class with both the variables and methods.

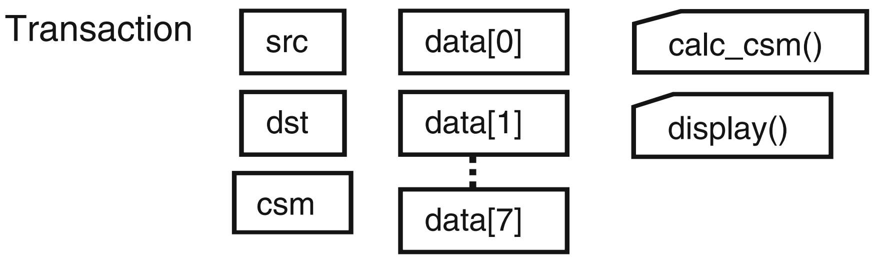

Fig. 8.2 Base `Transaction` class diagram

8.1.2 Extending the `Transaction` Class

Suppose you have a testbench that sends good transactions through the DUT and now you want to inject errors. If you follow the guidelines from Chapter 1, you would want to make as few code changes as possible to your existing testbench. So how can you reuse the existing `Transaction` class? Take the existing class and extend it to create a new class. This is done by declaring a new class, `BadTr`, as an extension of the current class. `Transaction` is the base class, and `BadTr` is the extended class. The code is shown in Sample 8.2 and in a diagram in Fig. 8.3.

Sample 8.2 Extended `Transaction` class

```
class BadTr extends Transaction;
  rand bit bad_csm;

  virtual function void calc_csm();
    super.calc_csm();            // Compute good csm
    if (bad_csm) csm = ~csm;     // Corrupt the csm bits
  endfunction

  virtual function void display(input string prefix="");
    $write("%sBadTr: bad_csm=%b, ", prefix, bad_csm);
    super.display();
  endfunction
```

Note that in Sample 8.2, the variable `csm` is does not need a hierarchical identifier. The `BadTr` class can see all the variables from the original `Transaction` plus its own variables such as `bad_csm`, as shown in Fig. 8.3. The `calc_csm` function in the extended class calls `calc_csm` in the base class using the `super` prefix. You can call a single level up, but going across multiple levels such as `super.super.new` is not allowed in SystemVerilog. This style, that reaches across multiple levels, would violate the rules of encapsulation by reaching across multiple boundaries.

The original `display` method printed a single line, starting with the prefix. So the extended `display` method prints the prefix, class name, and `bad_csm` with `$write` so the result is still on a single line.

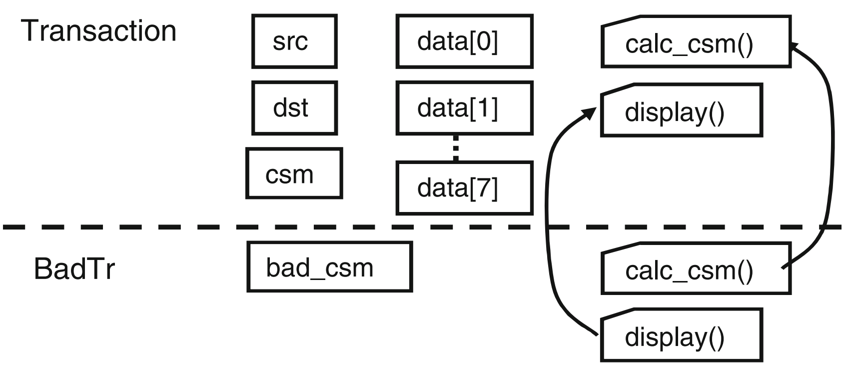

Fig. 8.3 Extended `Transaction` class diagram

Always declare methods inside a class as virtual so that they can be redefined in an extended class. This applies to all tasks and functions except the `new` function, which is called when the object is constructed, so there is no way to extend it. SystemVerilog always calls the `new` function based on the handle's type. Virtual methods are described fully in Section 8.3.2.

8.1.3 More OOP Terminology

Here is a quick glossary of terms. As explained in Chapter 5, the OOP term for a variable in a class is "property," and a task or function is called a "method." A base class is one that is not derived from any other class. When you extend a class, the original class (such as `Transaction`) is called the parent class or superclass. The extended class (`BadTr`) is also known as the derived or subclass. The "prototype" for a method is just the first line that shows the argument list and return type, if any. The prototype is used when you move the body of the method outside the class, but is needed to describe how the method communicates, as shown in Section 5.10.

8.1.4 Constructors in Extended Classes

When you start extending classes, there is one rule about constructors (`new` functions) to keep in mind. If your base class constructor has any arguments, the extended class must have a constructor and must call the base's constructor on its first line. In Sample 8.3, since `Base::new` has an argument, `Extended::new` must call it.

Sample 8.3 Constructor with arguments in an extended class

```
class Base;
  int val;
  function new(input int val); // Has an argument
    this.val = val;
  endfunction
endclass

class Extended extends Base;
  function new(input int val);
    super.new(val);  // Must be first line of new
    // Other constructor actions
  endfunction
endclass
```

8.1.5 Driver Class

The driver class in Sample 8.4 receives transactions from the generator and drives them into the DUT.

Sample 8.4 Driver class

```
class Driver;
 mailbox #(Transaction) gen2drv; // Mbx between Generator and here

  function new(input mailbox #(Transaction) gen2drv);
    this.gen2drv = gen2drv;
  endfunction

  virtual task run();
    Transaction tr;          // Handle to a Transaction object or
                             // a class extended from Transaction
    forever begin
      gen2drv.get(tr);      // Get transaction from generator
      tr.calc_csm();        // Process the transaction
      @ifc.cb;
      ifc.cb.src <= tr.src; // Send transaction
      ...
    end
  endtask
endclass
```

This class receives `Transaction` objects from the generator though the mailbox `gen2drv`, breaks them down into signal changes in the interface to stimulate the DUT. What happens if your generator instead sends a `BadTr` object into the class? OOP rules say that if you have a handle of the base type (`Transaction`), it can also point to an object of an extended type (`BadTr`). The handle `tr` can only reference things in the base class such as the variables `src`, `dst`, `csm`, and `data`, and the method `calc_csm`. So you can send `BadTr` objects into the driver without changing the `Driver` class.

See Chapter 10 and 11 for examples of fully functional drivers with advanced features such as virtual interfaces and callbacks.

When the driver calls `tr.calc_csm`, which one will be called, the one in `Transaction` or `BadTr`? Since `calc_csm` was declared as a virtual method in the base class in Sample 8.1, SystemVerilog chooses the proper method based on the type of object stored in `tr`. If the object is of type `Transaction`, SystemVerilog calls the task `Transaction::calc_csm`. If it is of type `BadTr`, SystemVerilog calls the function `BadTr::calc_csm`.

8.1.6 Simple Generator Class

The generator in Sample 8.5 for this testbench creates a random transaction and puts it in the mailbox to the driver. The following (bad) example shows how you might create the class from what you have learned so far. Note that this avoids a very common testbench bug by constructing a new transaction object every pass through the loop instead of just once outside. This bug is discussed in more detail in Section 7.6 on mailboxes.

Sample 8.5 Bad generator class

```
// Generator class that uses Transaction objects
// First attempt... too limited
class Generator;
  mailbox #(Transaction) gen2drv;// Carries transactions to driver
  Transaction tr;

  function new(input mailbox #(Transaction) gen2drv);
    this.gen2drv = gen2drv;    // this-> class-level var
  endfunction

  virtual task run(input int num_tr = 10);
    repeat (num_tr) begin
      tr = new();                        // Construct transaction
      `SV_RAND_CHECK(tr.randomize()); // Randomize it
      gen2drv.put(tr.copy());           // Send copy to driver
    end
  endtask
endclass
```

There is a big limitation with this generator. The `run` task constructs a transaction and immediately randomizes it. This means that the transaction uses whatever constraints are turned on by default. The only way you can change this would be to edit the `Transaction` class, which goes against the verification guidelines presented in this book. Worse yet, the generator only uses `Transaction` objects — there is no way to use an extended object such as `BadTr`. The fix is to separate the construction of `tr` from its randomization as shown below in Section 8.2.

As you build data-oriented classes such as network and bus transactions, you will see that they have common properties (`id`) and methods (`display`). Control-oriented classes such as the `Generator` and `Driver` classes also have a common structure. You can enforce this by making both of these classes extensions of a base `Transactor` class, with virtual methods for `run`, and `wrap_up`. Both the UVM and VMM has an extensive set of base classes for transactors, data, and much more.

8.2 Blueprint Pattern

A useful OOP technique is the "blueprint pattern." If you have a machine to make signs, you don't need to know the shape of every possible sign in advance. You just need a stamping machine and then change the die to cut different shapes. Likewise, when you want to build a transactor generator, you don't have to know how to build every type of transaction; you just need to be able to stamp new ones that are similar to a given transaction.

Instead of constructing and then immediately using an object, as in Sample 8.5, construct a blueprint object (the cutting die), and then modify its constraints with `constraint_mode`, or even replace it with an extended object, as shown in Fig. 8.4. Now when you randomize this blueprint, it will have the random values that you want. Make a copy of this object and send the copy to the downstream transactor.

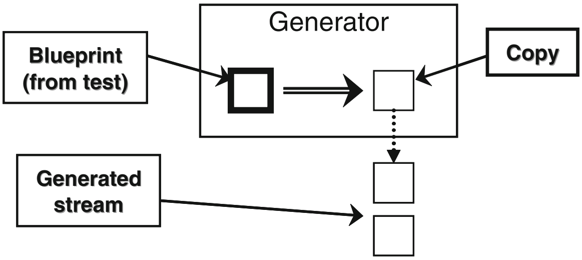

Fig. 8.4 Blueprint pattern generator

The beauty of this technique is that if you change the blueprint object, your generator creates an object of a different type. Using the sign analogy, you change the cutting die from a square to a triangle to make Yield signs, as shown in Fig. 8.5.

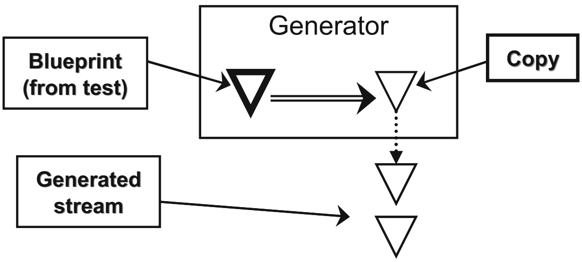

Fig. 8.5 Blueprint generator with new pattern

The blueprint is the "hook" that allows you to change the behavior of the generator class without having to change its code. You need to make a copy method that can make a copy of the blueprint to transmit, so that the original blueprint object is kept around for the next pass through the loop.

Sample 8.6 shows the generator class using the blueprint pattern. The important thing to notice is that the blueprint object is constructed in one place (the `new` function)

Sample 8.6 Generator class using blueprint pattern

```
class Generator;
  mailbox #(Transaction) gen2drv;
  Transaction blueprint;

  function new(input mailbox #(Transaction) gen2drv);
    this.gen2drv = gen2drv;
    blueprint = new();
  endfunction

  virtual task run(input int num_tr = 10);
    repeat(num_tr) begin
      `SV_RAND_CHECK(blueprint.randomize);
      gen2drv.put(blueprint.copy()); // Send copy to the driver
    end
  endtask
endclass
```

and used in another (the `run` task). Previous coding guidelines in this book said to separate the declaration and construction; similarly, you need to separate the construction and randomization of the blueprint object.

The `copy` method, which makes a duplicate of an object by copying its variables into a new object, is discussed in Sections 5.15 and 8.5. For now, remember that you must add it to the `Transaction` and `BadTr` classes. Sample 8.34 on page 304 shows an advanced generator using templates.

This generator constructs a new transaction every time the blueprint is randomized. This coding style prevents the classic OOP mailbox bug, as the mailbox will store handles to multiple unique objects, not that same single object.

Another advantage of randomizing the blueprint object over and over is that `randc` variables work correctly. The bad generator in Sample 8.5 constructed new objects every pass through the loop. Every object with a `randc` variable maintains a history of previous values generated for the variable. Every time you construct a new object, that history is lost, and the bad generator creates objects with separate `randc` variables. In Sample 8.6, only the blueprint object is randomized, so the `randc` history is maintained.

Section 8.2.3 shows how to change the blueprint.

8.2.1 The `Environment` Class

Chapter 1 discussed the three phases of execution: Build, Run, and Wrap-up. Sample 8.7 shows the environment class that instantiates all the testbench components, and runs these three phases. Also notice how the mailbox `gen2drv` carries transactions from the generator to the driver, and so is passed into the constructor for each.

Sample 8.7 `Environment` class

```
// Testbench environment class
class Environment;
  Generator gen;
  Driver drv;
  mailbox #(Transaction) gen2drv;

  virtual function void build();   // Build the environment by
    gen2drv = new();               // constructing the mailbox,
    gen = new(gen2drv);            // the generator,
    drv = new(gen2drv);            // and driver
  endfunction

  virtual task run();
    fork
      gen.run();
      drv.run();
    join
  endtask

  virtual task wrap_up();
    // Empty for now - call scoreboard for report
  endtask

endclass
```

8.2.2 A Simple Testbench

The test is contained in the top-level program shown in Sample 8.8. The basic test just lets the environment run with all the defaults.

Sample 8.8 Simple test program using environment defaults

```
program automatic test;

  Environment env;
  initial begin
    env = new();          // Construct the environment
    env.build();          // Build testbench objects
    env.run();            // Run the test
    env.wrap_up();        // Clean up afterwards
  end
endprogram
```

8.2.3 Using the Extended `Transaction` Class

To inject an error, you need to change the blueprint object from a `Transaction` object to a `BadTr`. You do this between the build and run phases in the environment. The top-level testbench in Sample 8.9 runs each phase of the environment and changes the blueprint. Note how all the references to `BadTr` are in this one file, so you don't have to change the `Environment` or `Generator` classes. You want to restrict the scope of where `BadTr` can be used, so a standalone `begin…end` block is used in the middle of the `initial` block. This makes a visually distinctive block of code. You can take a shortcut and construct the extended class in the declaration.

Sample 8.9 Injecting an extended transaction into testbench

```
program automatic test;

  Environment env;
  initial begin
    env = new();
    env.build();                  // Construct generator, etc.

    begin
      BadTr bad = new();          // Replace blueprint with
      env.gen.blueprint = bad;    // the "bad" one
    end

    env.run();                    // Run the test with BadTr
    env.wrap_up();                // Clean up afterwards
  end
endprogram
```

8.2.4 Changing Random Constraints with an Extended Class

In Chapter 6 you learned how to generate constrained random data. Most of your tests are going to need to further constrain the data, which is best done with inheritance. In Sample 8.10, the original Transaction class is extended to include a new constraint that keeps the destination address in the range of +/−100 of the source address.

Sample 8.10 replaces the generator's blueprint with an extended object that has an additional constraint. As you will learn later in this chapter, the `Nearby` class should have a `copy` method, but hold on for a few sections.

Sample 8.10 Adding a constraint with inheritance

```
class Nearby extends Transaction;
  constraint c_nearby {
    dst inside {[src-100:src+100]};
  }
  // copy method not shown
endclass


program automatic test;
  Environment env;
  initial begin
    env = new();
    env.build();                  // Construct generator, etc.

    begin
      Nearby nb = new();          // Create a new blueprint
      env.gen.blueprint = nb;     // Replace the blueprint
    end

    env.run();                    // Run the test with Nearby
    env.wrap_up();                // Clean up afterwards
  end
endprogram
```

Note that if you define a constraint in an extended class with the same name as one in the base class, the extended constraint replaces the base one. This allows you to change the behavior of existing constraints.

8.3 Downcasting and Virtual Methods

As you start to use inheritance to extend the functionality of classes, you need a few OOP techniques to control the objects and their functionality. In particular, a handle can refer to an object for a certain class, or any extended class. So what happens when a base handle points to an extended object? What happens when you call a method that exists in both the base and extended classes? This section explains what happens using several examples.

8.3.1 Downcasting with `$cast`

Downcasting or conversion is the act of casting a base class handle to point to an object that is a class extended from that base type. Consider the base and extended classes in Sample 8.11 and Fig. 8.6.

Sample 8.11 Base and extended class

```
class Transaction;
  rand bit [31:0] src;
  virtual function void display(input string prefix="");
    $display("%sTransaction: src=%0d", prefix, src);
  endfunction
endclass

class BadTr extends Transaction;
  bit bad_csm;
  virtual function void display(input string prefix="");
    $display("%sBadTr: bad_csm=%b", prefix, bad_csm);
    super.display(prefix);
  endfunction
endclass
```

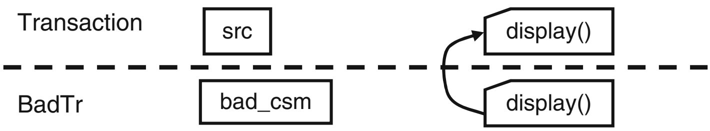

Fig. 8.6 Simplified extended transaction

You can assign an extended handle to a base handle, and no special code is needed, as shown in Sample 8.12. When a class is extended, all the base class variables and methods are included, so `src` is in the extended object. The assignment to `tr` is permitted, as any reference using the base handle `tr` is valid, such as `tr.src` and `tr.display`.

Sample 8.12 Copying extended handle to base handle

```
Transaction tr;
BadTr bad;
bad = new();         // Construct BadTr extended object
tr = bad;            // Base handle points to extended obj
                     // tr is downcast to point to BadTr type
$display(tr.src);    // Display variable in base class
tr.display;          // Calls BadTr::display
```

What if you try going in the opposite direction, copying a handle to a base object into an extended handle, as shown in Sample 8.13? This fails because the base object is missing properties that only exist in the extended class, such as `bad_csm`. The SystemVerilog compiler does a static check of the handle types and will not compile the second line.

Sample 8.13 Copying a base handle to an extended handle

```
tr = new();               // Construct base object
bad = tr;                 // ERROR: WILL NOT COMPILE
$display(bad.bad_csm);    // bad_csm is only in extended object
```

It is not always illegal to assign a base handle to an extended handle, but you must always use `$cast`. The assignment is allowed when the base handle points to an extended object. The `$cast` method checks the type of object referenced by the handles, not just the handle. If the source object is the same type as the destination, or a class extended from the destination's class, you can copy the address of the extended object from the base handle, `tr`, into the extended handle, `bad2`.

Sample 8.14 Using `$cast` to copy handles

```
Transaction tr;
BadTr bad, bad2;

bad = new();           // Construct BadTr extended object
tr = bad;              // Base handle points to extended object

// Check the object type & copy. Simulation error if mismatch
// If successful, bad2 points to the object referenced by tr
$cast(bad2, tr);

// Check for type mismatch, no simulation error
if($cast(bad2, tr))
  $display(bad2.bad_csm); // bad_csm exists in original object
else
  $display("ERROR: cannot assign tr to bad2");
```

When you use `$cast` as a task, SystemVerilog checks the type of the source object at run time and gives an error if it is not compatible with the destination. When you use `$cast` as a function, SystemVerilog still checks the type, but no longer prints an error if there is a mismatch. The `$cast` function returns zero when the types are incompatible, and one for compatible types.

As an alternative to the `if` statement in Sample 8.14, you could use something like the **`SV_RAND_CHECK`** macro from Section 6.3.2. You should not use an immediate `assert` statement as the assertion expression is not evaluated if you disable assertions, which means the `$cast` and `bad2` assignment will never execute.

8.3.2 Virtual Methods

By now you should be comfortable using handles with extended classes. What happens if you try to call a method using one of these handles? Sample 8.15 and 8.16 show base and extended classes and code that calls methods inside these classes.

Sample 8.15 `Transaction` and `BadTr` classes

```
class Transaction;
  rand bit [31:0] src, dst, data[8];  // Variables
  bit [31:0] csm;

  virtual function void calc_csm();   // XOR all fields
    csm = src ^ dst ^ data.xor;
  endfunction
endclass : Transaction

class BadTr extends Transaction;
  rand bit bad_csm;
  virtual function void calc_csm();
    super.calc_csm();                 // Compute good csm
    if (bad_csm) csm = ~csm;          // Corrupt the csm bits
  endfunction
endclass : BadTr
```

Sample 8.16 contains a block of code that uses handles of different types.

Sample 8.16 Calling class methods

```
Transaction tr;
BadTr bad;

initial begin
  tr = new();
  tr.calc_csm();     // Calls Transaction::calc_csm

  bad = new();
  bad.calc_csm();    // Calls BadTr::calc_csm

  tr = bad;          // Base handle points to ext obj
  tr.calc_csm();     // Calls BadTr::calc_csm
end
```

To decide which virtual method to call, SystemVerilog uses the object's type, not the handle's type. In the last statement of Sample 8.16, `tr` points to an extended object (`BadTr`) and so `BadTr::calc_csm` is called.

If you leave out the `virtual` modifier on `Transaction::calc_csm`, SystemVerilog checks the type of the handle `tr (Transaction)`, not the object. That last statement in Sample 8.16 calls `Transaction::calc_csm` – probably not what you wanted.

The OOP term for multiple methods sharing a common name is "polymorphism." It solves a problem similar to what computer architects faced when trying to make a processor that could address a large address space but had only a small amount of physical memory. They created the concept of virtual memory, where the code and data for a program could reside in memory or on a disk. At compile time, the program didn't know where its parts resided — that was all taken care of by the hardware plus operating system at run time. A virtual address could be mapped to some RAM chips, or the swap file on the disk. Programmers no longer needed to worry about this virtual memory mapping when they wrote code — they just knew that the processor would find the code and data at run time. See also Denning (2005).

8.3.3 Signatures and Polymorphism

There is a downside to using virtual methods: once you define one, all extended classes that define the same method must use the same "signature," i.e., the same number and type of arguments, plus return value, if any. You cannot add or remove an argument in an extended virtual method. This means you need to plan ahead.

There is a good reason that SystemVerilog and other OOP languages require that a virtual method must have the same signature as the one in the parent (or grandparent). If you were able to add an additional argument, or turn a task into a function, polymorphism would no longer work. Your code needs to be able to call a virtual method with the assurance that a method in a extended class will have the same interface.

8.3.4 Constructors are Never Virtual

When you call a virtual method, SystemVerilog checks the type of the object to decide if it should call the method in the base class or the extended. Now you can see why a constructor can not be virtual. When you call it, there is no object whose type can be checked. The object only exists after the constructor call starts.

8.4 Composition, Inheritance, and Alternatives

As you build up your testbench, you have to decide how to group related variables and methods together into classes. In Chapter 5 you learned how to build basic classes and include one class inside another. Previously in this chapter, you saw the basics of inheritance. This section shows you how to decide between the two styles, and also shows an alternative.

8.4.1 Deciding Between Composition and Inheritance

How should you tie together two related classes? Composition uses a "has-a" relationship. A packet has a header and a body. Inheritance uses an "is-a" relationship.

A `BadTr` is a `Transaction`, just with more information. Table 8.1 is a quick guide, with more detail below.

Table 8.1 Comparing inheritance to composition

Question	*Inheritance (is-a relationship)*	*Composition (has-a relationship)*
1. Do you need to group multiple extended classes together? (SystemVerilog does not support multiple inheritance)	No	Yes
2. Does the higher-level class represent objects at a similar level of abstraction?	Yes	No
3. Is the lower-level information always present or required?	Yes	No
4. Does the additional data need to remain attached to the original class while it is being processed by pre-existing code?	Yes	No

1. Are there several small classes that you want to combine into a larger class? For example, you may have a data class and header class and now want to make a packet class. SystemVerilog does not support multiple inheritance, where one class extends from several classes at once. Instead you have to use composition. Alternatively, you could extend one of the classes to be the new class, and manually add the information from the others.
2. In Sample 8.15, the `Transaction` and `BadTr` classes are both bus transactions created in a generator and driven into the DUT, so inheritance makes sense.
3. The lower-level information such as `src`, `dst`, and data must always be present for the Driver to send a transaction.
4. In Sample 8.15, the new `BadTr` class has a new field `bad_csm` and the extended `calc_csm` function. The `Generator` class just transmits a transaction and does not care about the additional information. If you use composition to create the error bus transaction, the `Generator` class would have to be rewritten to handle the new type.

If two objects seem to be related by both "is-a" and "has-a," you may need to break them down into smaller components.

8.4.2 Problems with Composition

The classical OOP approach to building a class hierarchy partitions functionality into small blocks that are easy to understand. However, testbenches are not standard software development projects, as was discussed in Section 5.16 on public vs. local attributes. Concepts such as information hiding (using local variables) conflict with building a testbench that needs maximum visibility and controllability. Similarly, dividing a transaction into smaller pieces may cause more problems than it solves.

When you are creating a class to represent a transaction, you may want to partition it to keep the code more manageable. For example, you may have an Ethernet MAC frame and your testbench uses two flavors, normal (`II`) and Virtual LAN (`VLAN`). Using composition, you could create a basic cell `EthMacFrame` with all the common fields such as `da` and `sa` and a discriminant variable, `kind`, to indicate the type as shown in Sample 8.17. There is a second class to hold the VLAN information, which is included in `EthMacFrame`.

Sample 8.17 Building an Ethernet frame with composition

```
// Not recommended
class EthMacFrame;
  typedef enum {II, IEEE} kind_e;
  rand kind_e kind;
  rand bit [47:0] da, sa;
  rand bit [15:0] len;

  ...
  rand Vlan vlan_h;
endclass

class Vlan;
  rand bit [15:0] vlan;
endclass
```

There are several problems with composition. First, it adds an extra layer of hierarchy, so you are constantly having to add an extra name to every reference. The VLAN information is called `eth_h.vlan_h.vlan`. If you start adding more layers, the hierarchical names become a burden.

A more subtle issue occurs when you want to instantiate and randomize the hierarchy of classes. What does the `EthMacFrame` constructor create? Since `kind` is random, you don't know whether to construct a `Vlan` object when `new` is called. When you randomize the class, the constraints set variables in both the `EthMacFrame` and `Vlan` objects based on the random `kind` field. You have a circular dependency in that randomization only works on objects that have been instantiated, but you can't instantiate these objects until `kind` has been chosen.

The only solution to the construction and randomization problems is to always instantiate all objects in `EthMacFrame::new`. However, if you are always using all alternatives, why divide the Ethernet cell into two different classes?

8.4.3 Problems with Inheritance

Inheritance can solve some of these issues. Variables in the extended classes can be referenced without the extra hierarchy as in `eth_h.vlan`. You don't need a discriminant, but you may find it easier to have one variable to test rather than doing type-checking as shown in Sample 8.18.

Sample 8.18 Building an Ethernet frame with inheritance

```
// Not recommended
class EthMacFrame;
  typedef enum {II, IEEE} kind_e;
  rand kind_e kind;
  rand bit [47:0] da, sa;
  rand bit [15:0] len;
  ...
endclass

class Vlan extends EthMacFrame;
  rand bit [15:0] vlan;
endclass
```

On the downside, a set of classes that use inheritance always requires more effort to design, build, and debug than a set of classes without inheritance. Your code must use `$cast` whenever you have an assignment from a base handle to an extended handle. Building a set of virtual methods can be challenging, as they all have to have the same signature. If you need an extra argument, you need to go back and edit the entire set, and possibly the method calls too.

There are also problems with randomization. How do you make a constraint that randomly chooses between the two kinds of frame and sets the proper variables? You can't put a constraint in `EthMacFrame` that references the `vlan` field.

The final issue is with multiple inheritance. In Fig. 8.7, you can see how the VLAN frame is extended from a normal MAC frame. The problem is that these different standards reconverged. SystemVerilog does not support multiple inheritance, so you could not create the VLAN / Snap / Control frame through inheritance.

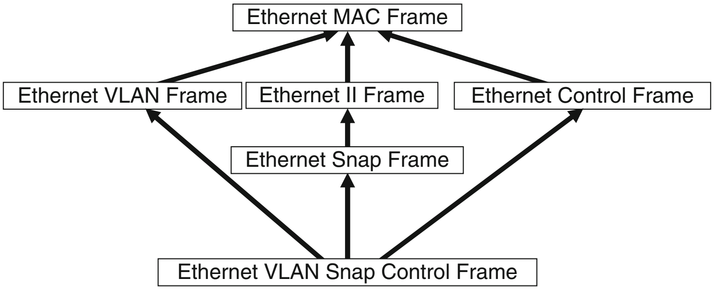

Fig. 8.7 Multiple inheritance problem

8.4.4 A Real-World Alternative

If composition leads to large hierarchies, but inheritance requires extra code and planning to deal with all the different classes, and both have difficult construction and randomization, what can you do? You can instead make a single, flat class that has all the variables and methods. This approach leads to a very large class, but it handles all the variants cleanly. You have to use the discriminant variable often to tell which variables are valid, as shown in Sample 8.19. It contains several conditional constraints, which apply in different cases, depending on the value of `kind`.

Sample 8.19 Building a flat Ethernet frame

```
class eth_mac_frame;
  typedef enum {II, IEEE} kind_e;
  rand kind_e kind;
  rand bit [47:0] da, sa;
  rand bit [15:0] len, vlan;
  rand bit [ 7:0] data[];
  ...
  constraint eth_mac_frame_II {
    if (kind == II) {
      data.size() inside {[46:1500]};
      len == data.size();
  }}
  constraint eth_mac_frame_ieee {
    if (kind == IEEE) {
      data.size() inside {[46:1500]};
      len < 1522;
  }}
endclass
```

Regardless of how you build your classes, define the typical behavior and constraints in the class, and then use inheritance to inject new behavior at the test level.

8.5 Copying an Object

In Sample 8.6, the generator first randomized, and then copied the blueprint to make a new transaction. Take a closer look at the `copy` function in Sample 8.20. Also see Section 5.15 for more examples of `copy` functions.

Sample 8.20 Base transaction class with a virtual `copy` function

```
class Transaction;
  rand bit [31:0] src, dst, data[8];  // Variables
  bit [31:0] csm;

  virtual function Transaction copy();
    copy = new();              // Construct destination object
    copy.src  = this.src;      // Copy data fields
    copy.dst  = this.dst;      // The prefix "this." is
    copy.data = this.data;     //   not needed, but makes code
    copy.csm  = this.csm;      //   more explicit
    return copy;               // Return handle to copy
  endfunction
endclass
```

When you extend the `Transaction` class to make the class `BadTr`, the `copy` function still has to return a `Transaction` object. This is because the extended virtual function must match the base `Transaction::copy`, including all arguments and return type, as shown in Sample 8.21

Sample 8.21 Extended transaction class with virtual `copy` method

```
class BadTr extends Transaction;
  rand bit bad_csm;

  virtual function Transaction copy();
    BadTr bad;
    bad = new();               // Construct extended object
    bad.src     = this.src;    // Copy data fields
    bad.dst     = this.dst;
    bad.data    = this.data;
    bad.csm     = this.csm;
    bad.bad_csm = this.bad_csm;
    return bad;                // Return handle to copy
  endfunction
endclass : BadTr
```

8.5.1 Specifying a Destination for Copy

The previous `copy` methods always constructed a new object. An improvement for `copy` is to specify the location where the copy should be put. This technique is useful when you want to reuse an existing object, and not allocate a new one.

Sample 8.22 Base transaction class with `copy` function

```
class Transaction;

  virtual function Transaction copy(input Transaction to=null);
    if (to == null)
      copy = new();          // Construct new object
    else
      copy = to;             // or use existing
    copy.src  = this.src;    // Copy data fields
    copy.dst  = this.dst;
    copy.data = this.data;
    copy.csm  = this.csm;
    return copy;
  endfunction
```

The only difference is the additional argument to specify the destination, and the code to test that a destination object was passed to this method. If nothing was passed (the default), construct a new object, or else use the existing one.

Since you have added a new argument to a virtual method in the base class, you will have to add it to the same method in the extended classes, such as `BadTr`.

Sample 8.23 Extended transaction class with new `copy` function

```
class BadTr extends Transaction;

  virtual function Transaction copy(input Transaction to=null);
    BadTr bad;
    if (to == null)
      bad = new();                  // Create a new object
    else
      $cast(bad, to);               // Reuse existing one
    super.copy(bad);                // Copy base data fields
    bad.bad_csm = this.bad_csm;     // Copy extended fields
    return bad;
  endfunction
endclass : BadTr
```

Notice how `BadTr::copy` only needs to copy the fields in the extended class and can use the base class method, `Transaction::copy` to copy its own fields.

8.6 Abstract Classes and Pure Virtual Methods

By now you have seen classes with methods to perform common operations such as copying and displaying. One goal of verification is to create code that can be shared across multiple projects. If your company standardizes on a common set of classes and methods, it is easier to reuse code between projects.

OOP languages such as SystemVerilog have two constructs to allow you to build a shareable base class. The first is an abstract class, which is a class that can be extended, but not instantiated directly. It is defined with the `virtual` keyword. The second is a pure virtual method, which is a prototype without a body. A class extended from an abstract class can only be instantiated if all pure virtual methods have bodies. The `pure` keyword specifies that a method declaration is a prototype, and not just an empty virtual method. A pure method has no `endfunction` or `endtask`. Lastly, pure virtual methods can only be declared in an abstract class. An abstract class can contain pure virtual methods, virtual methods with and without a body, and non-virtual methods. Note that if you define a virtual method without a body, i.e. no code inside, you can call it but it just immediately returns.

Sample 8.24 shows an abstract class, `BaseTr`, which is a base class for transactions. It starts with a some useful properties such as `id` and `count`. The constructor makes sure every instance has a unique ID. Next are pure virtual methods to compare, copy, and display the object.

Sample 8.24 Abstract class with pure virtual methods

```
virtual class BaseTr;
  static int count;     // Number of instance created
  int id;               // Unique transaction id

  function new();
    id = count++;       // Give each object a unique ID
  endfunction

  pure virtual function bit compare(input BaseTr to);
  pure virtual function BaseTr copy(input BaseTr to=null);
  pure virtual function void display(input string prefix="");

endclass : BaseTr
```

You can declare handles of type `BaseTr`, but you cannot construct objects of this type. You need to extend the class and provide implementations for all the pure virtual methods.

Sample 8.25 shows the definition of the `Transaction` class, which has been extended from `BaseTr`. Since `Transaction` has bodies for all the pure virtual methods extended from `BaseTr`, you can construct objects of this type in your testbench.

Sample 8.25 Transaction class extends abstract class

```
class Transaction extends BaseTr;
  rand bit [31:0] src, dst, csm, data[8];

  extern function new();
  extern virtual function bit compare(input BaseTr to);
  extern virtual function BaseTr copy(input BaseTr to=null);
  extern virtual function void display(input string prefix="");

endclass

function Transaction::new();
  super.new();
endfunction : new

function bit Transaction::compare(input BaseTr to);
  Transaction tr;
  if(!$cast(tr, to))  // Check if 'to' is correct type
    $finish;
  return ((this.src  == tr.src) &&
          (this.dst  == tr.dst) &&
          (this.csm  == tr.csm) &&
          (this.data == tr.data));
endfunction : compare

function BaseTr Transaction::copy(input BaseTr to=null);
  Transaction cp;
  if (to == null) cp = new();
  else            $cast(cp, to);
  cp.src  = this.src;    // Copy the data fields
  cp.dst  = this.dst;
  cp.data = this.data;
  cp.csm  = this.csm;
  return cp;
endfunction : copy

function void Transaction::display(input string prefix="");
  $display("%sTransaction %0d src=%h, dst=%x, csm=%x",
           prefix, id, src, dst, csm);
endfunction : display
```

Abstract classes and pure virtual methods let you build testbenches that have a common look and feel. This allows any engineer to read your code and quickly understand the structure.

8.7 Callbacks

One of the main guidelines of this book is to create a single verification environment that you can use for all tests with no changes. The key requirement is that this testbench must provide a "hook" where the test program can inject new code without modifying the original classes. Your driver may want to do the following.

- Inject errors
- Drop the transaction
- Delay the transaction
- Synchronize this transaction with others
- Put the transaction in the scoreboard
- Gather functional coverage data

Rather than try to anticipate every possible error, delay, or disturbance in the flow of transactions, the driver just needs to "call back" a method that is defined in the top-level test. The beauty of this technique is that the callback method can be defined differently in every test. As a result, the test can add new functionality to the driver using callbacks, without editing the `Driver` class. For some drastic behaviors such as dropping a transaction, you need to code this in the class ahead of time, but this is a known pattern. The reason why the transaction is dropped is left to the callback.

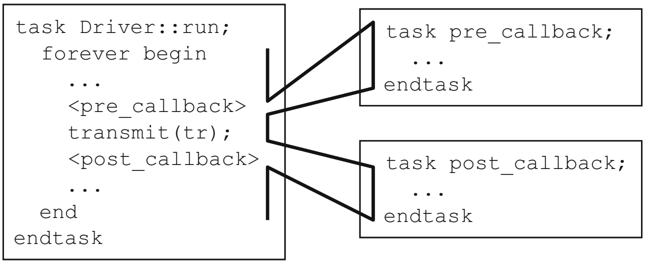

Fig. 8.8 Callback flow

In Fig. 8.8, the `Driver::run` task loops forever with a call to a `transmit` task. Before sending the transaction, `run` calls the pre-transmit callback, if any. After sending the transaction, it calls the post-callback task, if any. By default, there are no callbacks, so `run` just calls `transmit`.

You could make `Driver::run` a virtual method and then override its behavior in an extended class, perhaps `MyDriver::run`. The drawback to this is that you might have to duplicate all the original method's code in the new method if you are injecting new behavior. Now if you made a change in the base class, you would have to remember to propagate it to all the extended classes. Additionally, you can inject a callback without modifying the code that constructed the original object.

8.7.1 Creating a Callback

A callback task is created in the top-level test and called from the driver, the lowest level of the environment. However, the driver does not have to have any knowledge of the test – it just has to use a generic class that the test can extend. The driver in Sample 8.27 uses a queue to hold the callback objects, which allows you to add multiple objects. The base callback class in Sample 8.26 is an abstract class that must be extended before being used. Your callback is a task so it can have delays.

Sample 8.26 Base callback class

```
virtual class Driver_cbs;  // Driver callbacks

  virtual task pre_tx(ref Transaction tr, ref bit drop);
    // By default, callback does nothing
  endtask

  virtual task post_tx(ref Transaction tr);
    // By default, callback does nothing
  endtask
endclass
```

Sample 8.27 Driver class with callbacks

```
// Partial example - see Sample 8-4 for more details
class Driver;
  Driver_cbs cbs[$];    // Queue of callback objects

  task run();
    bit drop;
    Transaction tr;

    forever begin
      drop = 0;
      agt2drv.get(tr);  // Agent to driver mailbox
      foreach (cbs[i]) cbs[i].pre_tx(tr, drop);
      if (drop) continue;
      transmit(tr);     // Actual work
      foreach (cbs[i]) cbs[i].post_tx(tr);
    end
  endtask
endclass
```

Note that while Driver_cbs is an abstract class, pre_tx and post_tx are not pure virtual methods. This is because a typical callback uses only one of them. If a class has even one pure virtual method without an implementation, OOP rules won't allow you to instantiate it.

Callbacks are part of both VMM and UVM. This callback technique is not related to Verilog PLI callbacks or SVA callbacks.

8.7.2 Using a Callback to Inject Disturbances

A common use for a callback is to inject some disturbance such as causing an error or delay. The testbench in Sample 8.28 randomly drops packets using a callback object. Callbacks can also be used to send data to the scoreboard or to gather functional coverage values. Note that you can put callback objects in the queue with the push_back() or push_front() depending on the order in which you want these to be called. For example, you probably want the scoreboard called after any tasks that may delay, corrupt, or drop a transaction. You should only gather coverage after a transaction has been successfully transmitted.

Sample 8.28 Test using a callback for error injection

```
class Driver_cbs_drop extends Driver_cbs;
 virtual task pre_tx(ref Transaction tr, ref bit drop);
    // Randomly drop 1 out of every 100 transactions
    drop = ($urandom_range(0,99) == 0);
  endtask
endclass


program automatic test;
  Environment env;

  initial begin
    env = new();
    env.gen_cfg();
    env.build();

    begin              // Create error injection callback
      Driver_cbs_drop dcd = new();
      env.drv.cbs.push_back(dcd); // Put into driver's Q
    end

    env.run();
    env.wrap_up();
  end

endprogram
```

8.7.3 A Quick Introduction to Scoreboards

The design of your scoreboard depends on the design under test. A DUT that processes atomic transactions such as packets may have a scoreboard that contains a transform function to turn the input transactions into expected values, a memory to hold these values, and a compare method. A processor design needs a reference model to predict the expected output, and the comparison between expected and actual values may happen at the end of simulation.

Sample 8.29 shows a simple scoreboard that stores transactions in a queue of expected values. The first method saves an expected transaction, and the second tries to find an expected transaction that matches an actual one that was received by the testbench. Note that when you search through a queue, you can get 0 matches (transaction not found), 1 match (ideal case) or multiple matches (you need to do a more sophisticated match).

Sample 8.29 Simple scoreboard for atomic transactions

```
class Scoreboard;
  Transaction scb[$];  // Store expected tr's in queue

  function void save_expected(input Transaction tr);
    scb.push_back(tr);
  endfunction

  function void compare_actual(input Transaction tr);
    int q[$];

    q = scb.find_index(x) with (x.src == tr.src);
    case (q.size())
      0: $display("No match found");
      1: scb.delete(q[0]);
      default:
        $display("Error, multiple matches found!");
    endcase
  endfunction : compare_actual
endclass : Scoreboard
```

8.7.4 Connecting to the Scoreboard with a Callback

The testbench in Sample 8.30 creates its own extension of the driver's callback class and adds a reference to the driver's callback queue. Note that the scoreboard callback needs a handle to the scoreboard so it can call the method to save the expected transaction. This example does not show the monitor side, which will need its own callback to send the actual transaction to the scoreboard for comparison.

Sample 8.30 Test using callback for scoreboard

```
class Driver_cbs_scoreboard extends Driver_cbs;
  Scoreboard scb;

 virtual task pre_tx(ref Transaction tr, ref bit drop);
    // Put transaction in the scoreboard
    scb.save_expected(tr);
  endtask

  function new(input Scoreboard scb);
    this.scb = scb;
  endfunction
endclass


program automatic test;
  Environment env;

  initial begin
    env = new();
    env.gen_cfg();
    env.build();

    begin                        // Create scoreboard callback
      Driver_cbs_scoreboard dcs = new(env.scb);
      env.drv.cbs.push_back(dcs); // Put into driver's Q
    end

    env.run();
    env.wrap_up();
  end

endprogram
```

The VMM recommends that you use callbacks for scoreboards and functional coverage. The monitor transactor can use a callback to compare received transactions with expected ones. The monitor callback is also the perfect place to gather functional coverage on transactions that are actually sent by the DUT.

You may have thought of putting the scoreboard or functional coverage group in a transactor, and connect it to the testbench using a mailbox. This is a poor solution for several reasons. These testbench components are almost always passive and asynchronous, so they only wake up when the testbench has data for them, plus they never pass information to a downstream transactor. Thus a transactor that has to monitor multiple mailboxes concurrently is an overly complex solution. Additionally, you may sample data from several points in your testbench, but a transactor is designed for a single source. Instead, put methods in your scoreboard and coverage classes to gather data, and connect them to the testbench with callbacks.

The UVM recommends a TLM analysis port for connecting monitors / drivers to scoreboards and functional coverage. A description of this construct is beyond the scope of this book, but you can think of it as a mailbox with an optional consumer.

8.7.5 Using a Callback to Debug a Transactor

If a transactor with callbacks is not working as expected, you can add a debug callback. You can start by adding a callback to display the transaction. If there are multiple instances of the transactor, create a unique identifier for each. Put debug code before and after the other callbacks to locate the one that is causing the problem. Even for debug, you want to avoid making changes to the testbench environment.

8.8 Parameterized Classes

As you become more comfortable with classes, you may notice that a class, such as a stack or generator, only works on a single data type. This section shows how you can define a single parameterized class that works with multiple data types.

8.8.1 A Simple Stack

A common data structure is a stack, which has `push` and `pop` methods to store and retrieve data. Sample 8.31 shows a simple stack that works with the `int` data type.

Sample 8.31 Stack using the int type

```
parameter int SIZE = 100;
class IntStack;
  local int stack[SIZE];           // Holds data values
  local int top;

  function void push(input int i); // Push value on top
    stack[top++] = i;
  endfunction : push

  function int pop();              // Remove value from top
    return stack[--top];
  endfunction

endclass : IntStack
```

The problem with this class is that it only works with integers. If you want to make a stack for real numbers, you would have to copy the class, and change the data type from `int` to `real`. This quickly leads to a proliferation of classes, which can become a maintenance problem if you ever want to add new operations such as traversing or printing the stack contents.

In SystemVerilog you can add a data type parameter to a class and then specify a type when you declare handles to that class. This is similar to, but more powerful than, a parameterized module, where you can specify a value such as bus width when it is instantiated. SystemVerilog's parameterized classes are similar to templates in C++.

Sample 8.32 is a parameterized class for a stack. Notice how the type `T` is defined on the first line with a default type of `int`.

Sample 8.32 Parameterized class for a stack

```
parameter int SIZE = 100;
class Stack #(type T=int);
  local T stack[SIZE];           // Holds data values
  local int top;

  function void push(input T i);  // Push new value on top
    stack[top++] = i;
  endfunction : push

  function T pop();              // Remove value from top
    return stack[--top];
  endfunction

endclass : Stack
```

The step of specifying values to a parameterized class is called specialization. Sample 8.33 declares a handle to the stack class with a real data type.

Sample 8.33 Creating the parameterized stack class

```
initial begin
  Stack #(real) rStack;          // Specialize the stack class

  rStack = new();                // Construct a stack of reals
  for(int i=0; i<SIZE; i++)
    rStack.push(i*2.0);          // Push values onto stack

  for(int i=0; i<SIZE; i++)
    $display("%f", rStack.pop()); // Pop values off stack
end
```

Generators are a great example of a class that can be parameterized. Once you have defined the class for one, the same structure works for any data type. Sample 8.34 takes the atomic generator from Sample 8.6 and adds a parameter so you can generate any random object. The generator should be part of a package of verification classes. It needs to specify a the default type, so it uses `BaseTr` from Sample 8.24 as this abstract class should also be part of the verification package.

Sample 8.34 Parameterized generator class using blueprint pattern

```
class Generator #(type T=BaseTr);
  mailbox #(Transaction) gen2drv;
  T blueprint;                      // Blueprint object

  function new(input mailbox #(Transaction) gen2drv);
    this.gen2drv = gen2drv;
    blueprint = new();              // Create default
  endfunction

  task run(input int num_tr = 10);
    T tr;
    repeat (num_tr) begin
      `SV_RAND_CHECK(blueprint.randomize);
      $cast(tr, blueprint.copy()); // Make a copy
      gen2drv.put(tr);              // Send to driver
    end
  endtask
endclass
```

Using the `Transaction` class from Sample 8.25 and the generator in Sample 8.34, you can build a simple testbench like in Sample 8.35. It starts the generator and prints the first five transactions, using the mailbox synchronization shown in Sample 7.40.

Sample 8.35 Simple testbench using parameterized generator class

```
program automatic test;

  initial begin
    Generator #(Transaction) gen;
    mailbox #(Transaction) gen2drv;
    gen2drv = new(1);
    gen = new(gen2drv);

    fork
      gen.run();

      repeat (5) begin
        Transaction tr;
        gen2drv.peek(tr);  // Get next transaction
        tr.display();
        gen2drv.get(tr);   // Remove transaction
      end

    join_any
  end
endprogram // test
```

8.8.2 Sharing Parameterized Classes

When you specialize a parameterized class, as in the `real` stack in Sample 8.33, you are creating a new data type, with no OOP relationship to any other specialization. For example, you can not use `$cast()` to convert between a stack of real variables and one of integers. For that, you need a common base class as shown in Sample 8.36.

Sample 8.36 Common base class for parameterized generator class

```
class GenBase;
endclass

class Generator #(type T=BaseTr) extends GenBase;
  // See Generator class in Sample 8-34
endclass

  GenBase gen_queue[$];
  Generator #(Transaction) gen_good;
  Generator #(BadTr)       gen_bad;

  initial begin
    gen_good = new();                  // Construct good generator
    gen_queue.push_back(gen_good);     // Save it in the queue
    gen_bad = new();                   // Construct bad generator
    gen_queue.push_back(gen_bad);      // Save it in the same queue
  end
```

Upcoming sections show more examples of parameterized classes.

8.8.3 Parameterized Class Suggestions

When creating parameterized classes, you should start with a non-parameterized class, debug it thoroughly, and then add parameters. This separation reduces your debug effort.

A common set of virtual methods in your transaction class help you when creating parameterized classes. The `Generator` class uses the `copy` method, knowing that it always has the same signature. Likewise, the `display` method allows you to easily debug transactions as they flow through your testbench components.

The system functions `$typename()` and `$bits()` are helpful when your class needs to know the name and width of the parameter. The `$typename(T)` function returns the name of the parameter type such as `int`, `real`, or the class name for a handle. The `$bits()` function returns the width of the parameter. For complex types such as structures and arrays, it returns the number of bits required to hold an expression as a bit stream. The UVM transaction print methods use this function to get the fields to line up correctly.

Macros are an alternative to parameterized classes. For example, you could define a macro for the generator and pass it the transaction data type. Macros are harder to debug than parameterized classes, unless your compiler outputs the expanded code.

If you need to define several related classes that all share the same transaction type, you could use parameterized classes or a single large macro. In the end, how you define your classes is not as important as what goes into them.

8.9 Static and Singleton Classes

This section and the next show advanced OOP concepts that are used extensively in the UVM and VMM. You could try to understand UVM's factory mechanism by reading the source code with its many methods, but this section should save you several days of experimentation with a greatly simplified example. This chapter shows several alternatives so you can understand why the UVM did not pick a more simple alternative.

One of the goals of OOP is to eliminate global variables and methods as the resulting code is hard to maintain and reuse. Their names exist in the global name space, potentially causing name space collisions. Does `packet_count` refer to TCP/IP packets or some other protocol? Instead, put a variable called `count` in the `Packet` class to avoid any ambiguity.

8.9.1 Dynamic Class to Print Messages

Sometimes, however, you really need globals. For example, all verification methodologies provide a print service so you can filter messages and count errors. If you try to build such a class with what you have learned so far, it might look something like Sample 8.37.

Sample 8.37 Dynamic print class with static variables

```
class Print;
  static bit [31:0] error_count = 0, error_limit = -1;
  string class_name, instance_name;

  function new(input string class_name, instance_name);
    this.class_name    = class_name;
    this.instance_name = instance_name;
  endfunction

  function void error(input string ID, input string message);
    $display("@%0t %m [%s-%s] [%s] %s",
             $realtime, class_name, instance_name, ID, message);
    if (++error_count >= error_limit) begin
      $display("FATAL: Maximum error limit reached");

      $finish;
    end
  endfunction

endclass
```

This is a greatly simplified version of the VMM log class. The VMM code allows you to filter messages by the class and instance names, and many other features.

Sample 8.38 has a class that prints an error message with the `Print` class from Sample 8.37.

Sample 8.38 Transactor class with dynamic print object

```
class Xactor;
  Print p;
  function new();
    p = new("Xactor", "solo");
  endfunction // new

  task run();
    p.error("NYI", "This Xactor is not yet implemented");
  endtask
endclass // Xactor
```

The biggest limitation for the `Print` class is that every component in your testbench needs to instantiate it. The simple `Print` class above has a small footprint, but a realistic one, like VMM's, could have many strings and arrays, consuming a significant amount of memory. This overhead, when added to a transactor class might not be significant, but could overwhelm a small transaction class, such as an ATM cell, which only has 53 bytes.

8.9.2 Singleton Class to Print Messages

An alternative to constructing all these print objects is to not construct any. As you saw in section 5.11.4, you could declare the methods in the `Print` class to be static. These methods can only reference static variables, as shown in Sample 8.39.

Sample 8.39 Static print class

```
class Print;
  static bit [31:0] error_count = 0, error_limit = -1;
  static function void error(input string ID,
                              input string message);
    $display("@%0t %m [%s] %s", $realtime, ID, message);
    error_count++;
    if (error_count >= error_limit) begin
      $display("Maximum error limit reached");
      $finish;
    end
  endfunction
endclass
```

Now that the class is static, you can no longer have per-instance information such as the parent class's name and instance. Any filtering has to be based on other criteria.

Sample 8.40 Transactor class with static print class

```
class Xactor;
  task run();
    Print::error("NYI", "This Xactor is not yet implemented");
  endtask
endclass
```

Sample 8.40 shows the call to the `error()` method using the `Print` class name.

This style of class is known as a singleton class, as there is only one copy, the one allocated at elaboration time with the static variables.

As your static classes, such as the one in Sample 8.39, grow larger, you have to label everything with the `static` keyword, a small annoyance. Next, the class is allocated before simulation time, even if you never use it. Additionally, there is no handle to this class, so you can not pass it around your testbench. The alternative to a static class is a singleton class (or singleton pattern) with a single instance, which is a non-static class that is only constructed once. They are more difficult to create initially, but they can simplify your program's architecture. Many of the UVM's classes are singletons.

The singleton pattern is implemented by creating a class with a method that creates a new instance of the class if one does not exist. If an instance already exists, it simply returns a handle to that object. To make sure that the object cannot be instantiated any other way, you must make the constructor `protected`. Don't make it `local`, because an extended class might need to access the constructor.

8.9.3 Configuration Database with Static Parameterized Class

Another good use for static classes in verification is a database of configuration parameters. At the start of simulation you randomize the configuration of your system. In a small system, you can simply store these in a single class or hierarchy of classes and pass them around the testbench as needed. At some point though, this becomes too complicated as handles are passed up and down the hierarchy. Instead, create a global database of parameters, indexed by a name, that you can access anywhere in the testbench. UVM 1.0 introduced this concept, which is the basis for the following set of examples. This code has a single string index into the database, while a real database such as UVM's could have a property name, instance name, and other values. You could concatenate these to create a more complex index string.

One issue with a database is that you need to store values of different types, such as bit vectors, integers, real numbers, enumerated values, string, class handles, virtual interfaces, and more in a single database. While you could find a few common types such as bit vectors and a common base class, there are some type such as virtual interfaces that are unique, so there is no easy way to store them in a common database. Earlier versions of OVM and UVM recommended creating a class wrapper around virtual interfaces, but this required extra coding and was a common source of bugs.

What if you made a different database for each data type? You could use an associative array indexed by the parameter name. A real database might also have an instance name, but for this simple example, you can just concatenate all the names together to make a single index. Sample 8.41 shows the code for an integer database made from global methods.

Sample 8.41 Configuration database with global methods

```
int db_int[string];
function void db_int_set(input string name, input int value);
  db_int[name] = value;
endfunction

function void db_int_get(input string name, ref int value);
  value = db_int[name];
endfunction

function void db_int_print();
  foreach (db_int[i])
    $display("db_int[%s] = %0d", i, db_int[i]);
endfunction
```

You can generalize this into a parameterized class with the concepts from Section 8.8, as shown in Sample 8.42.

Sample 8.42 Configuration database with parameterized class

```
class config_db #(type T=int);
  T db[string];
  function void set(input string name, input T value);
    db[name] = value;
  endfunction

  function void get(input string name, ref T value);
    value = db[name];
  endfunction

  function void print();
    $display("Configuration database %s", $typename(T));
    foreach (db[i])
      $display("db[%s] = %p", i, db[i]);
  endfunction
endclass
```

You can now construct objects for an integer database, a real database, etc. The final problem is that each instance of the database is local to the scope where this class is instantiated. The solution shown in Sample 8.43 is to go global and make this a static class, that is a class with static properties and methods.

Sample 8.43 Configuration database with static parameterized class

```
class config_db #(type T=int);
  static T db[string];
  static function void set(input string name, input T value);
    db[name] = value;
  endfunction

  static function void get(input string name, ref T value);
    value = db[name];
  endfunction

  static function void print();
    $display("\nConfiguration database %s", $typename(T));
    foreach (db[i])
      $display("db[%s] = %0p", i, db[i]);
  endfunction
endclass
```

You can test the above code with Sample 8.44 and see how the parameterized class creates a new database for each type.

Sample 8.44 Testbench for configuration database

```
class Tiny;
   int i;
endclass // Tiny

int i = 42, j = 43, k;                    // Integers for database
real pi = 22.0/7.0, r;                   // Reals for database
Tiny t;                                  // Handle for database

initial begin
  config_db#(int)::set("i", i);          // Save an int in db
  config_db#(int)::set("j", j);          // Save another
  config_db#(real)::set("pi", pi);       // Save a real in db

  t = new();
  t.i = 8;
  config_db#(Tiny)::set("t", t);         // Save a handle in db
  config_db#(Tiny)::set("null", null); // Test null handles

  config_db#(int)::get("i", k);           // Fetch an int
  $display("Fetched value (%0d) of i (%0d) ", i, k);

  config_db#(int)::print();               // Print int db
  config_db#(real)::print();              // Print real db
  config_db#(Tiny)::print();              // Print Tiny db
end
```

With singletons implemented as single instances instead of static class members, you can initialize the singleton lazily, creating it only when it is needed.

The UVM database allows wildcards and other regular expressions, which requires a more complex lookup scheme than associative arrays.

8.10 Creating a Test Registry

In a real design, compiling your test and DUT takes a significant amount of time. If you want to run 100 tests, each in a separate program block, you need to recompile before each test, 100 times in all. This is a waste of CPU time as most of the code has not changed. If you make 100 program blocks, each with a single test, and connect all these programs in the model, you then need a way to disable all but one program block. The best solution is to include all tests and testbenches inside one program block, compile this once with the DUT. This section shows how you can select one test per run with a Verilog command line switch.

8.10.1 Test registry with Static Methods

Earlier examples in this book have a program that contains one test. For this new approach, each test is a separate class, all which are in a single program block, either imported from a package or included at compile time. The test classes are constructed, registered in a test registry, and then, at run time, you can choose the desired test at runtime. This follows an early VMM style.

First you need a base test class that your tests can extend from. Sample 8.45 shows an abstract class that contains a handle for the Environment class and a pure virtual task that is a placeholder for the method that contains your test code.

Sample 8.45 Base test class

```
virtual class TestBase;
  Environment env;
  pure virtual task run_test();
  function new();
    env = new();
  endfunction
endclass
```

The core of the test registry class is an associative array of handles to all the tests, indexed by the test name. The `TestRegistry` class, shown in Sample 8.46, is a static class with only static variables and methods, and is never constructed. The `get_test()` method reads the Verilog command line argument to determine which test to execute.

Sample 8.46 Test registry class

```
class TestRegistry;
  static TestBase registry[string];

  static function void register(string name, TestBase t);
    registry[name] = t;
  endfunction // register

  static function TestBase get_test();
    string name;
    if (!$value$plusargs("TESTNAME=%s", name))
      $display("ERROR: No +TESTNAME switch found");
    return registry[name];
  endfunction
endclass // TestRegistry
```

Sample 8.47 show how you can extend `TestBase` to create a simple test that runs all the environment phases. The last line of the example is a declaration that calls the constructor, which also registers the test. All the test objects are constructed, but only one is run.

Sample 8.47 Simple test in a class

```
// Repeat for each test
class TestSimple extends TestBase;

  function new();
    env = new();
    TestRegistry::register("TestSimple", this);
  endfunction

  virtual task run_test();
    $display("%m");
    env.gen_config();
    env.build();
    env.run();
    env.wrap_up();
  endtask
endclass

TestSimple TestSimple_handle = new();  // Needed for each class
```

The program in Sample 8.48 now just asks the test registry for a test object and runs it. The test classes can be declared in a package and imported, or declared inside or outside the program block.

Sample 8.48 Program block for test classes

```
program automatic test;
  TestBase tb;
  initial begin
    tb = TestRegistry::get_test();
    tb.run_test();
  end
endprogram
```

Sample 8.49 shows how you can create a test class that injects new behavior by changing the generator's blueprint to create bad transactions.

Sample 8.49 Test class that puts a bad transaction in the generator

```
class TestBad extends TestBase;
  function new();
    env = new();
    TestRegistry::register("TestBad", this);
  endfunction // new

  virtual task run_test();
    $display("%m");
    env.gen_config();
    env.build();
    begin
      BadTr bad = new();
      env.gen.blueprint = bad;
    end
    env.run();
    env.wrap_up();
  endtask
endclass

TestBad TestBad_handle = new(); // Declaration & constructing
```

This short example allows you to compile many tests into a single simulation executable and choose your test at runtime, saving many recompiles. This pattern is fine when you are starting out with a handful of tests, but the next section shows more powerful approach.

8.10.2 Test Registry with a Proxy Class

The previous section's test registry works well for smaller test environments, but has several limitations for real projects. First, you need to remember to construct every test class, otherwise the registry can not locate it. Second, every test gets constructed at the start of simulation, even though only one is actually run.

When verifying a large design, there could be hundreds of tests, so constructing all of them wastes valuable simulation time and memory.

Consider this analogy. When you are looking to buy a car, you can go to a dealer to see the choices. If there are only a few variants, white or black, with or without sunroof, the dealer can stock one of each model with little overhead. This is what you saw in the previous section, where the test registry had an object of each test type.

What if there are many different models, each in one of a dozen colors, with variants such as radios, sunroofs, air conditioning, sports packages, and engines? The dealer could never have one of each type on his lot as there are hundreds of combinations. Instead he would show you a catalog with all the choices. You pick the options that you want, and the factory builds one to your specification. Likewise, the test registry can have a lot of small classes, each which knows how to build a complete test. The small class has low overhead, so even a thousand objects would not consume much memory. Now when you want to run test N, imagine flipping through the catalog (test registry) until you find a picture of your test, and you then tell the factory to build an object of that type.

The test registry needs a table (analogous to the above catalog) that goes from test names to objects. In section 8.10.1, this table is an associative array of `TestBase` handles, indexed by a string, shown in Sample 8.46. What if instead, you had a parameterized class whose only job is to construct a test? The UVM uses a design pattern called a proxy class whose only role is to build the actual desired class. The proxy class is lightweight in that it only contains a few properties and methods, and thus consumes little memory or CPU time. It acts like the picture in the car dealer's catalog, holding a representation of what you can build.

The next few code samples show how the UVM class factory works. Because the code in this book is a simplified version of the real UVM classes, the name has been changed to SVM, SystemVerilog Methodology, so that you won't confuse it with the real thing. Hopefully you will find this explanation of a simple factory easier to understand than trying to read the UVM source code.

First is Sample 8.50 which has the common base class from which everything else is built. It is a abstract class because you should never construct an object of this type, only classes extended from this one.

Sample 8.50 Common SVM base class

```
virtual class svm_object;
  // Empty class
endclass
```

Next is the component class in Sample 8.51. In the UVM, a component is a time-consuming object that forms the testbench hierarchy, similar to a VMM transactor. In this simplified example, the hierarchical parent handle has been removed.

Sample 8.51 Component class

```
virtual class svm_component extends svm_object;
  protected svm_component m_children[string];
  string name;

  function new(string name);
    this.name = name;
    $display("%m name='%s'", name);
  endfunction

  pure virtual task run_test();
endclass
```

Now define `svm_object_wrapper`, the abstract common base class for the proxy class as shown in Sample 8.52. It has pure virtual methods to return the name of the class type, and create an object of this type.

Sample 8.52 Common base class for proxy class

```
virtual class svm_object_wrapper;
  pure virtual function string get_type_name();
  pure virtual function svm_object create_object(string name);
endclass
```

Now for the crucial class, **`svm_component_registry`** shown in Sample 8.53. This is a lightweight class that can be constructed with little overhead. It is parameterized with the test class type and name. Once you have an instance of this class, your testbench can construct the actual test class at any time, using the **`create_object`** method. This is a singleton class as you only need one copy to create an instance of the test class. At the start of simulation, the static handle **`me`** is initialized by calling the **`get()`** method that constructs the first instance if needed.

Sample 8.53 Parameterized proxy class

```
class svm_component_registry #(type T=svm_component,
                               string Tname="<unknown>")
        extends svm_object_wrapper;

  typedef svm_component_registry #(T,Tname) this_type;

  virtual function string get_type_name();
    return Tname;
  endfunction

  local static this_type me = get();   // Handle to singleton

  static function this_type get();
    if (me == null) begin               // Is there an instance?
      svm_factory f = svm_factory::get(); // Build factory
      me = new();                        // Build the singleton
      f.register(me);                    // Register class
    end
    return me;
  endfunction

  virtual function svm_object create_object (string name="");
    T obj;
    obj = new(name);
    return obj;
  endfunction

  static function T create(string name);
    create = new(name);
  endfunction

endclass : svm_component_registry
```

The last major class is **svm_factory**, which, at its core, is just a singleton class that holds the array, **m_type_names**, to go from test case name to the proxy class that creates an instance of the test class. Also in this class in Sample 8.54 is the `get_test` method that reads the test name from the simulation run command line and constructs an instance of the test class. Unlike Sample 8.46, you even get a little self checking.

Sample 8.54 Factory class

```
class svm_factory;
  // Assoc array from string to svm_object_wrapper handle
  static svm_object_wrapper m_type_names[string];

  static svm_factory m_inst;  // Handle to this singleton

  static function svm_factory get();
    if (m_inst == null) m_inst = new();
    return m_inst;
  endfunction

  static function void register(svm_object_wrapper c);
    m_type_names[c.get_type_name()] = c;
  endfunction

  static function svm_component get_test();
    string name;
    svm_object_wrapper test_wrapper;
    svm_component test_comp;

    if (!$value$plusargs("SVM_TESTNAME=%s", name)) begin
      $display("FATAL +SVM_TESTNAME not found");
      $finish;
    end
    $display("%m found +SVM_TESTNAME=%s", name);
    test_wrapper = svm_factory::m_type_names[name];
    $cast (test_comp, test_wrapper.create_object(name));
    return test_comp;
  endfunction
endclass : svm_factory
```

Lastly is a base test class, extended from `svm_component` shown in Sample 8.55. It uses the macro **`svm_component_utils`** to define a new data type, `type_id`, that points to the proxy class. The macro stringifies the token `T` that holds the class name, and turns it into a string containing the value of `T` with the syntax: **`` `"T`" ``**.

Sample 8.55 Base test class and registration macro

```
`define svm_component_utils(T) \
  typedef svm_component_registry #(T,`"T`") type_id; \
  virtual function string get_type_name (); \
    return `"T`"; \
  endfunction

class TestBase extends svm_component;
  Environment env;
  `svm_component_utils(TestBase)

  function new(string name);
    super.new(name);
    $display("%m");
    env = new();
  endfunction

  virtual task run_test();
  endtask
endclass : TestBase
```

Sample 8.56 Test program

```
program automatic test;
  initial begin
    svm_component test_obj;
    test_obj = svm_factory::get_test();
    test_obj.run_test();
  end
endprogram
```

Here are the steps that happen when you start a simulation with the command line switch `+SVM_TESTNAME=TestBase`.

- With the macro `svm_component_utils`, the class `TestBase` defines the type `type_id` based on the class `svm_component_registry`, with the parameters `TestBase` and `"TestBase"`. Because this is a new type, the simulator initializes the static variable `svm_component_registry::me` by calling the `get` method that instantiates the class. This instance is registered in the factory. What does all this mean? There is now an object that can construct the `TestBase` class, and you can get to it through the factory.
- Simulation now starts and the factory's `get_test` method reads the test name from the command line. This string is used an index into the registry to get a handle to the proxy object. This object's `create_object` method constructs an instance of the `TestBase` object.

- The program calls the test object's `run_test` method, which calls the steps for the specific class. Now the TestBase class in Sample 8.55 does not do anything interesting, but add a call to `svm_component_utils` macro to the test classes in Sample 8.47 and Sample 8.49 and you can run tests.

Now you can see the basic UVM flow to start tests. The registry contains a list of proxy classes that can construct test objects.

8.10.3 UVM Factory Build

The UVM factory can also construct objects for any class in the testbench with the `create` method in Sample 8.53. Sample 8.57 show how to build a driver.

Sample 8.57 UVM factory build example

```
driver drv;
drv = driver::type_id::create("drv", this);
```

The above code calls the static method `create` to construct an object of type `driver`. In UVM, the second argument points to the parent of the component being created.

The UVM factory allows you to override the component so that when you build a component, you get an extended one instead.

You may have noticed a change in terminology. In classic OOP, you "construct" a class by calling the `new` method, based on the handle type and assigning the address to the handle on the left side of the assignment statement. With the UVM factory pattern, you "build" an object by calling the static `create` method. This could make an object of the same type as the handle, or an extended type.

8.11 Conclusion

The software concept of inheritance, where new functionality is added to an existing class, parallels the hardware practice of extending the design's features for each generation, while still maintaining backwards compatibility.

For example, you can upgrade your PC by adding a larger capacity disk. As long as it uses the same interface as the old one, you do not have to replace any other part of the system, yet the overall functionality is improved.

Likewise, you can create a new test by "upgrading" the existing driver class to inject errors. If you use an existing callback in the driver, you do not have to change any of the testbench infrastructure.

You need to plan ahead if you want use these OOP techniques. By using virtual methods and providing sufficient callback points, your test can modify the behavior of the testbench without changing its code. The result is a robust testbench that does not need to anticipate every type of disturbance (error-injection, delays, synchronization) that you may want as long as you leave a hook where the test can inject its own behavior.

The testbench is more complex than what you have previously constructed, but there is a payback in that the tests become smaller and easier to write. The testbench does the hard work of sending stimulus and checking responses, so the test only has to make small tweaks to cause specialized behavior. An extra few lines of testbench code might replace code that would have to be repeated in every single test.

Lastly, OOP techniques improve your productivity by allowing you to reuse classes. For example, a parameterized class for a stack that operates on any other class, rather than a single type, saves you from having to create duplicate code.

8.12 Exercises

1. Given the following class, create a method in an extended class `ExtBinary` that multiplies `val1` and `val2` and returns an integer.

```
class Binary;
  rand bit [3:0] val1, val2;

  function new(input bit [3:0] val1, val2);
    this.val1 = val1;
    this.val2 = val2;
  endfunction

  virtual function void print_int(input int val);
     $display("val=0d%0d", val);
  endfunction

endclass
```

2. Starting with the solution to Exercise 1, use the `ExtBinary` class to initialize `val1=15`, `val2=8`, and print out the multiplied value.

3. Starting with the solution to Exercise 1, create an extended class `Exercise3` that constrains `val1` and `val2` to be less than 10.

4. Starting with the solution to Exercise 3, use the `Exercise3` class to randomize `val1` and `val2`, and print out the multiplied value.

5. Given the class in Exercise 1, and the following declarations, and an extended class `ExtBinary`, what will handles `mc`, `mc2`, and `b` point to after executing each code snippet a-d, or will a compile error occur?

```
Binary b;
ExtBinary mc, mc2;
```

a.
```
mc = new(15,8);
b = mc;
```
b.
```
b = new(15, 8);
mc = b;
```
c.
```
mc = new(15, 8);
b = mc;
mc2 = b;
```
d.
```
mc = new(15, 8);
b = mc;
if($cast(mc2, b))
   $display("Success");
else
  $display("Error: cannot assign");
```

6. Given the classes `Binary` and `ExtBinary` in Exercise 1 and the following copy function for class `Binary`, create the function `ExtBinary::copy`.

```
virtual function Binary Binary::copy();
  copy = new(15,8);
  copy.val1 = val1;
  copy.val2 = val2;
endfunction
```

7. From the solution to Exercise 6, use the copy function to copy the object pointed to by the extended class handle `mc` to the extended class handle `mc2`.

8. Using code Sample 8.26 to Sample 8.28 in Section 8.7.1 and 8.7.2 of the text, add the ability to randomly delay a transaction between 0 and 100ns.

9. Create a class that can compare any data type using the case equality operators, `===` and `!==`. It contains a compare function that returns a 1 if the two values match, 0 otherwise. By default it compares two 4-bit data types.

10. Using the solution from Exercise 9, use the comparator class to compare two 4-bit values, `expected_4bit` and `actual_4bit`. Next, compare two values of type `color_t`, `expected_color` and `actual_color`. Increment an error counter if an error occurs.