Chapter 5
Basic OOP

5.1 Introduction

With procedural programming languages such as Verilog and C, there is a strong division between data structures and the code that uses them. The declarations and types of data are often in a different file than the algorithms that manipulate them. As a result, it can be difficult to understand the functionality of a program, as the two halves are separate.

Verilog users have it even worse than C users, as there are no structures in Verilog, only bit vectors and arrays. If you wanted to store information about a bus transaction, you would need multiple arrays: one for the address, one for the data, one for the command, and more. Information about transaction N is spread across all the arrays. Your code to create, transmit, and receive transactions is in a module that may or may not be actually connected to the bus. Worst of all, the arrays are all static, so if your testbench only allocated 100 array entries, and the current test needed 101, you would have to edit the source code to change the size and recompile. As a result, the arrays are sized to hold the greatest conceivable number of transactions, but during a normal test, most of that memory is wasted.

Object-Oriented Programming (OOP) lets you create complex data types and tie them together with the routines that work with them. You can create testbenches and system-level models at a more abstract level by calling routines to perform an action rather than toggling bits. When you work with transactions instead of signal transitions, you are working at a higher level, and your code is more easily written and understood. As a bonus, your testbench is decoupled from the design details, making it more robust and easier to maintain and reuse on future projects.

C. Spear and G. Tumbush, *SystemVerilog for Verification: A Guide to Learning the Testbench Language Features*, DOI 10.1007/978-1-4614-0715-7_5,

If you already are familiar with OOP, skim this chapter, as SystemVerilog follows OOP guidelines fairly closely. Be sure to read Section 5.18 to learn how to build a testbench. Chapter 8 presents advanced OOP concepts such as inheritance and more testbench techniques; it should be read by everyone.

5.2 Think of Nouns, not Verbs

Grouping data and code together helps you in creating and maintaining large testbenches. How should data and code be brought together? You can start by thinking of how you would perform the testbench's job.

The goal of a testbench is to apply stimulus to a design and then check the result to see if it is correct. The data that flows into and out of the design is grouped together into transactions such as bus cycles, opcodes, packets, or data samples. The best way to organize the testbench is around the transactions, and the operations that you perform on them. In OOP, the transaction is the focus of your testbench.

You can think of an analogy between cars and testbenches. Early cars required detailed knowledge about their internals (nouns) to operate. You had to advance or retard the spark, open and close the choke, keep an eye on the engine speed and be aware of the traction of the tires if you drove on a slippery surface such as a wet road. Today your interactions with the car are at a high level. When you get into a car, you perform discrete actions (verbs), such as starting, moving forward, turning, stopping, and listening to music while you drive. If you want to start a car, just turn the key in the ignition, and you are done. Get the car moving by pressing the gas pedal; stop it with the brakes. Are you driving on snow? Don't worry: the anti-lock brakes help you stop safely and in a straight line. You don't have to think about the low level details.

Your testbench should be structured the same way. Traditional testbenches were oriented around the operations that had to happen: create a transaction, transmit it, receive it, check it, and make a report. Instead, you should think about the structure of the testbench, and what each part does. The generator creates transactions and passes them to the next level. The driver talks with the design that responds with transactions that are received by a monitor. The scoreboard checks these against the expected data. You should divide your testbench into blocks, and then define how they communicate. This chapter shows many examples of these components.

How do you represent these blocks in SystemVerilog? A class may describe a data-centric block such as a bus transaction, network packet, or CPU instruction. Or a class might represent a control block such as a driver or scoreboard. Either way, a class encapsulates the data together with the routines that manipulate it. The details of how the class implements actions such as data generation or checking is hidden from the outside, making the class more reusable.

5.3 Your First Class

Sample 5.1 shows a class for a generic transaction. It contains an address, a checksum, and an array of data values. There are two routines in the `Transaction` class: one that displays the address, and another that computes a checksum of the data.

To make it easier to match the beginning and end of a named block, you can put a label on the end of it. In Sample 5.1 these end labels may look redundant, but in complex code, with many nested blocks, the labels help you find the mate for a simple `end`, `endtask`, `endfunction`, or `endclass`.

Sample 5.1 Simple transaction class

```
class Transaction;
  bit [31:0] addr, csm, data[8];

  function void display();
    $display("Transaction: %h", addr);
  endfunction : display

  function void calc_csm();
    csm = addr ^ data.xor;
  endfunction : calc_csm

endclass : Transaction
```

Every company has its own naming style. This book uses the following convention: Class names start with a capital letter and avoid using underscores, as in `Transaction` or `Packet`. Constants are all upper case, as in `CELL_SIZE`, and variables are lower case, as in `count` or `opcode`. You are free to use whatever style you want.

5.4 Where to Define a Class

You can define and use classes in SystemVerilog in a `program`, `module`, `package`, or outside of any of these.

When you start a project, you might store a single class per file. When the number of files gets too large, you can group a set of related classes and type definitions into a SystemVerilog `package` as shown in Sample 5.2. For instance, you might group together all USB3 transactions and BFMs into a single `package`. Now you can compile the package separately from the rest of the system. Unrelated classes, such as those for other transactions, scoreboards, or different protocols, should remain in separate files.

Code samples in this book leave out the packages to keep the text more compact.

Sample 5.2 Class in a package

```
// File abc.svh
package abc;
  class Transaction;
    // Class body
  endclass
endpackage
```

Sample 5.3 shows how to import a package into a program.

Sample 5.3 Importing a package in a program

```
program automatic test;
  import abc::*;
  Transaction tr;

  // Test code

endprogram
```

5.5 OOP Terminology

What separates you, an OOP novice, from an expert? The first thing is the words you use. You already know some OOP concepts from working with Verilog. Here are some OOP terms, definitions, and rough equivalents in Verilog 2001.

- Class – a basic building block containing routines and variables. The analogue in Verilog is a module.
- Object – an instance of a class. In Verilog, you need to instantiate a module to use it.
- Handle – a pointer to an object. In Verilog, you use the name of an instance when you refer to signals, and routines from outside the module. A handle is like the address of the object, but stored in a pointer that can only refer to one type. A handle is similar to a reference in other OOP languages.
- Property – a variable that holds data. In Verilog, this is a signal such as a register or wire.
- Method – the procedural code that manipulates variables, contained in tasks and functions. Verilog modules have tasks and functions plus `initial` and `always` blocks.
- Prototype – the header of a routine that shows the name, type, and argument list plus return type. The body of the routine contains the executable code. See Section 5.10 for more on prototypes and out-of-body experiences.

This book uses the more traditional terms from Verilog of "variable" and "routine" when discussing non-OOP code, and "property" and "method" for classes.

In Verilog you build complex designs by creating modules and instantiating them hierarchically. In OOP you create classes and construct them (creating objects) to create a similar hierarchy. Modules are instantiated during compilation while classes are constructed at run time.

Here is an analogy to explain these OOP terms. Think of a class as the blueprint for a house. This plan describes the structure of the house, but you cannot live in a blueprint; you need to build the physical house. An object is the actual house. Just as one set of blueprints can be used to build a whole subdivision of houses, a single class can be used to build many objects. The house address is like a handle in that it uniquely identifies your house. Inside your house you have things such as lights (on or off), with switches to control them. A class has variables that hold values, and routines that control the values. A class for the house might have many lights. A single call to `turn_on_porch_light()` sets the light variable ON in a single house.

5.6 Creating New Objects

Both Verilog and OOP have the concept of instantiation, but there are some differences in the details. A Verilog module, such as a counter, is instantiated when you compile your design. A SystemVerilog class, such as a network packet, is instantiated during simulation, when needed by the testbench. Verilog instances are static, as the hardware does not change during simulation; only signal values change. SystemVerilog stimulus objects are constantly being created and used to drive the DUT and check the results. Later, the objects may be freed so their memory can be used by new ones. Back to the house analogy: the address is normally static, unless your house burns down, causing you to construct a new one. Garbage collection at home is not automatic, especially if you have teenagers.

The analogy between OOP and Verilog has a few other exceptions. The top-level Verilog module is implicitly instantiated. However, a SystemVerilog class must be instantiated before it can be used. Next, a Verilog instance name only refers to a single instance, whereas a SystemVerilog handle can refer to many objects, though only one at a time.

5.6.1 Handles and Constructing Objects

In Sample 5.4, `tr` is a handle that points to an object of type `Transaction`. For brevity, you can just say `tr` is a `Transaction` handle.

Sample 5.4 Declaring and using a handle

```
Transaction tr;  // Declare a handle
tr = new();      // Allocate a Transaction object
```

When you declare the handle `tr`, it is initialized to the special value `null`. On the next line, call the `new()` function to construct the `Transaction` object.

This special `new` function allocates space for the `Transaction`, initializes the variables to their default value (0 for 2-state variables and X for 4-state ones), and returns the address where the object is stored. For example, the `Transaction` class has two 32-bit registers (`addr` and `csm`) and an array with eight values (data), for a total of 10 longwords, or 40 bytes. So when you call `new`, SystemVerilog allocates at least 40 bytes of storage. If you have used C, this step is similar to calling the `malloc` function. Note that SystemVerilog requires additional memory for 4-state variables and housekeeping information such as the object's type.

This process is called instantiation as you are making an instance of the class. The `new` function is sometimes called the constructor, as it builds the object, just as your carpenter constructs a house from wood and nails. For every class, SystemVerilog creates a default `new` function to allocate and initialize an object.

5.6.2 *Custom Constructor*

You can define your own `new()` function to set your own values. Note that you must not give a return value type as the constructor is a special function and automatically returns a handle to an object of the same type as the class.

Sample 5.5 Simple user-defined new() function

```
class Transaction;
  logic [31:0] addr, csm, data[8];

  function new();
    addr = 3;
    data = '{default:5};
  endfunction

endclass
```

In Sample 5.5, first SystemVerilog allocates the space for the object automatically. Next it sets `addr` and `data` to fixed values but leaves `csm` at its default value of X. You can use arguments with default values to make a more flexible constructor, as shown in Sample 5.6. Now you can specify the value for `addr` and `data` when you call the constructor, or use the default values.

Sample 5.6 A new() function with arguments

```
class Transaction;
  logic [31:0] addr, csm, data[8];

  function new(input logic [31:0] a=3, d=5);
    addr = a;
    data = '{default:d};
  endfunction
endclass

initial begin
  Transaction tr;
  tr = new(.a(10));  // a=10, d uses default of 5
end
```

How does SystemVerilog know which `new()` function to call? It looks at the type of the handle on the left side of the assignment. In Sample 5.7, the call to `new` inside the `Driver` constructor calls the `new()` function for `Transaction`, even though the one for `Driver` is closer. Since `tr` is a `Transaction` handle, SystemVerilog does the right thing and creates an object of type `Transaction`.

Sample 5.7 Calling the right new() function

```
class Transaction;
  logic [31:0] addr, csm, data[8];
endclass : Transaction

class Driver;
  Transaction tr;
  function new();       // Driver's new function
    tr = new();         // Call the Transaction new function
  endfunction
endclass : Driver
```

5.6.3 Separating the Declaration and Construction

You should avoid declaring a handle and calling the constructor, `new`, all in one statement. While this is legal syntax and less verbose, it can create ordering problems, as the constructor is called before the first procedural statement. You might need to initialize objects in a certain order, but if you call `new()` in the declaration, you won't have the same control. Additionally, if you forget to use `automatic` storage, the constructor is called at the start of simulation, not when the block is entered.

5.6.4 The Difference Between New() and New[]

You may have noticed that this `new()` function looks a lot like the `new[]` operator described in Section 2.3, used to set the size of dynamic arrays. They both allocate memory and initialize values. The big difference is that the `new()` function is called to construct a single object, whereas the `new[]` operator is building an array with multiple elements. `new()` can take arguments for setting object values, whereas `new[]` only takes a single value for the number of elements in the array. Just remember that the new with square brackets `[]` is for arrays, while the one with parentheses `()` is for classes, which usually contain methods.

5.6.5 Getting a Handle on Objects

New OOP users often confuse an object with its handle. The two are very distinct. You **declare** a handle and **construct** an object. Over the course of a simulation, a handle can point to many objects. This is the dynamic nature of OOP and SystemVerilog. Don't get the handle confused with the object.

In Sample 5.8, `t1` first points to one object, then another. Fig. 5.1 shows the resulting handles and objects.

Sample 5.8 Allocating multiple objects

```
Transaction t1, t2;  // Declare two handles
t1 = new();          // Allocate first Transaction object
t2 = t1;             // t1 & t2 point to it
t1 = new();          // Allocate second Transaction object
```

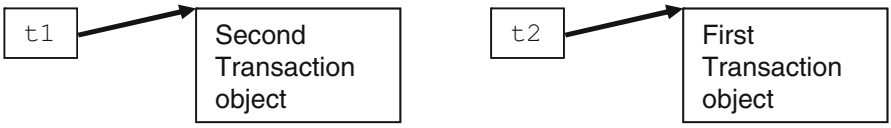

Fig. 5.1 Handles and objects after allocating multiple objects

Why would you want to create objects dynamically? During a simulation you may need to create hundreds or thousands of transactions. SystemVerilog lets you create objects automatically, when you need them. In Verilog, you would have to use a fixed-size array large enough to hold the maximum number of transactions.

Note that this dynamic creation of objects is different from anything else offered before in the Verilog language. An instance of a Verilog module and its name are bound together statically during compilation. Even with `automatic` variables, which come and go during simulation, the name and storage are always tied together.

An analogy for handles is people who are attending a conference. Each person is similar to an object. When you arrive, a badge is "constructed" by writing your name on it. This badge is a handle that can be used by the organizers to keep track of each person. When you take a seat for the lecture, space is allocated. You may have multiple badges for attendee, presenter, or organizer. When you leave the conference, your badge may be reused by writing a new name on it, just as a handle can point to different objects through assignment. Lastly, if you lose your badge and there is nothing to identify you, you will be asked to leave. The space you take, your seat, is reclaimed for use by someone else.

5.7 Object Deallocation

Now you know how to create an object — but how do you get rid of it? For example, your testbench creates and sends thousands of transactions, such as packets, instructions, frames, interrupts, etc. into your DUT. Once you know the transaction has completed successfully, you don't need to keep it around. You should reclaim the memory; otherwise, a long simulation might run out of memory.

Garbage collection is the process of automatically freeing objects that are no longer referenced. One way SystemVerilog can tell if an object is no longer being used is by keeping track of the number of handles that point to it. When the last handle no longer references an object, SystemVerilog releases the memory for it. (The actual algorithm to find unused objects varies between simulators. This section describes reference counting, which is the easiest to understand).

Sample 5.9 Creating multiple objects

```
Transaction t;  // Create a handle
t = new();      // Allocate a new Transaction
t = new();      // Allocate a second one, free the first
t = null;       // Deallocate the second
```

The second line in Sample 5.9 calls `new()` to construct an object and store the address in the handle `t`. The next call to `new()` constructs a second object and stores its address in `t`, overwriting the previous value. Since there are no handles pointing to the first object, SystemVerilog can deallocate it. The object may be deleted immediately, or after a short wait. The last line explicitly clears the handle so that now the second object can be deallocated.

If you are familiar with C++, these concepts of objects and handles are familiar, but there are some important differences. A SystemVerilog handle can only point to objects of one type, so they are called "type-safe." In C, a typical void pointer is only an address in memory, and you can set it to any value or modify it with operators such as pre-increment. You cannot be sure that a pointer is valid. A C++ typed pointer is much safer, but you may be tempted by C's flexibility. SystemVerilog does not

allow any modification of a handle or using a handle of one type to refer to an object of another type. (SystemVerilog's OOP specification is closer to Java than C++).

Since SystemVerilog garbage collects an object when no more handles refer to it, you can be sure your code always uses valid handles. In C / C++, a pointer can refer to an object that no longer exists. Garbage collection in those languages is manual, so your code can suffer from "memory leaks" when you forget to deallocate objects.

SystemVerilog cannot garbage collect an object that is still referenced somewhere by a handle. For example, if you keep objects in a linked list, SystemVerilog cannot deallocate the objects until you manually clear all handles by setting them to `null`. If an object contains a routine that forks off a thread, the object is not deallocated while the thread is running. Likewise, any objects that are used by a spawned thread may not be deallocated until the thread terminates. See Chapter 7 for more information on threads.

5.8 Using Objects

Now that you have allocated an object, how do you use it? Going back to the Verilog module analogy, you can refer to variables and routines in an object with the "." notation, as shown in Sample 5.10.

Sample 5.10 Using variables and routines in an object

```
Transaction t;      // Declare a handle to a Transaction
t = new();          // Construct a Transaction object
t.addr = 32'h42;    // Set the value of a variable
t.display();        // Call a routine
```

In strict OOP, the only access to variables in an object should be through accessor functions such as `get()` and `put()`. This is because accessing variables directly limits your ability to change the underlying implementation in the future. If a better (or simply different) algorithm comes along in the future, you may not be able to adopt it because you would also need to modify all of the references to the variables.

The problem with this methodology is that it was written for large software applications with lifetimes of a decade or more. With dozens of programmers making modifications, stability is paramount. However, you are creating a testbench, where the goal is maximum control of all variables to generate the widest range of stimulus values. One of the ways to accomplish this is with constrained-random stimulus generation, which cannot be done if a variable is hidden behind a screen of methods. While the `get()` and `put()` methods are fine for compilers, GUIs, and APIs, you should stick with public variables that can be directly accessed anywhere in your testbench.

The exception to this rule is for verification IP that is created and maintained by a group such as a company that has no direct relationship to the end user. For example, if you purchase a PCI transactor from another company, they will restrict access to the internals, forcing you to treat it as a black box. The developer must give you enough methods to generate both good transactions and inject all flavors of errors.

5.9 Class Methods

A method in a class is just a `task` or `function` defined inside the scope of the class. Sample 5.11 defines `display()` methods for the `Transaction` and `PCI_Tran`. SystemVerilog calls the correct one, based on the handle type.

Sample 5.11 Routines in the class

```
class Transaction;
  bit [31:0] addr, csm, data[8];
  function void display();
    $display("@%0t: TR addr=%h, csm=%h, data=%p",
             $time, addr, csm, data);
  endfunction
endclass

class PCI_Tran;
  bit [31:0] addr, data;  // Use realistic names
  function void display();
    $display("@%0t: PCI: addr=%h, data=%h", $time, addr, data);
  endfunction
endclass

Transaction t;
PCI_Tran pc;

initial begin
  t = new();      // Construct a Transaction
  t.display();    // Display a Transaction
  pc = new();     // Construct a PCI transaction
  pc.display();   // Display a PCI Transaction
end
```

A method in a class uses automatic storage by default, so you don't have to worry about remembering the `automatic` modifier.

5.10 Defining Methods Outside of the Class

A good rule of thumb is you should limit a piece of code to one "page" or screen in your favorite editor to keep it understandable. You may be familiar with this rule for routines, but it also applies to classes. If you can see everything in a class on the screen at one time, it is easier to understand.

However, if each method takes a page, how can the whole class fit on a page? In SystemVerilog you can break a method into the prototype (method name and arguments) inside the class, and the body (the procedural code) that goes after the class.

Here is how you create out-of-block declarations. Copy the first line of the method, with the name and arguments, and add the `extern` keyword at the beginning. Now take the entire method and move it after the class body, and add the class name and two colons (`::` the scope operator) before the method name. The above classes could be defined as shown in Sample 5.12.

Sample 5.12 Out-of-block method declarations

```
class Transaction;
  bit [31:0] addr, csm, data[8];
  extern function void display();
endclass

function void Transaction::display();
  $display("@%0t: Transaction addr=%h, csm=%h, data=%p",
           $time, addr, csm, data);
endfunction

class PCI_Tran;
  bit [31:0] addr, data;  // Use realistic names
  extern function void display();
endclass

function void PCI_Tran::display();
  $display("@%0t: PCI: addr=%h, data=%h",
           $time, addr, data);
endfunction
```

A common coding mistake is when the prototype does not match the out-of-body. SystemVerilog requires that the prototype be identical to the out-of-block method declaration, except for the class name and scope operator, `::`. The prototype can have qualifiers such as `local`, `protected`, or `virtual`, but not the out-of-body. If any arguments have default values, they must be given in the prototype, but they are optional in the out-of-body.

Another common mistake is to leave out the class name when you declare the method outside of the class. As a result, it is defined at the next higher scope (probably the program or package scope), and the compiler gives an error when the task tries to access class-level variables and methods. This is shown in Sample 5.13.

Sample 5.13 Out-of-body method missing class name

```
class Bad_OOB;
  bit [31:0] addr, csm, data[8]; // Class-level variable
  extern function void display();
endclass

function void display();          // Missing "Bad_OOB::"
  $display("addr=%0d", addr);     // Error, addr not found
endfunction
```

5.11 Static Variables vs. Global Variables

Every object has its own local variables that are not shared with any other object. If you have two `Transaction` objects, each has its own `addr`, `csm`, and `data` variables. Sometimes though, you need a variable that is shared by all objects of a certain type. For example, you might want to keep a running count of the number of transactions that have been created. Without OOP, you would probably create a global variable. Then you would have a global variable that is used by one small piece of code, but is visible to the entire testbench. This "pollutes" the global name space and makes variables visible to everyone, even if you want to keep them local.

5.11.1 A Simple Static Variable

In SystemVerilog you can create a static variable inside a class. This variable is shared amongst all instances of the class, but its scope is limited to the class. In Sample 5.14, the static variable `count` holds the number of objects created so far. It is initialized to 0 in the declaration because there are no transactions at the beginning of the simulation. Each time a new object is constructed, it is tagged with a unique value, and `count` is incremented.

Sample 5.14 Class with a static variable

```
class Transaction;
  static int count = 0; // Number of objects created
  int id;               // Unique instance ID
  function new();
    id = count++;       // Set ID, bump count
  endfunction
endclass

Transaction t1, t2;
initial begin
  t1 = new();           // 1st instance, id=0, count=1
  $display("First id=%0d, count=%0d", t1.id, t1.count);
  t2 = new();           // 2nd instance, id=1, count=2
  $display("Second id=%0d, count=%0d", t2.id, t2.count);
end
```

In Sample 5.14, there is only one copy of the static variable `count`, regardless of how many `Transaction` objects are created. You can think that `count` is stored with the class and not the object. The variable `id` is not static, so every `Transaction` has its own copy, as shown in Fig. 5.2. Now you don't need to make a global variable for the count.

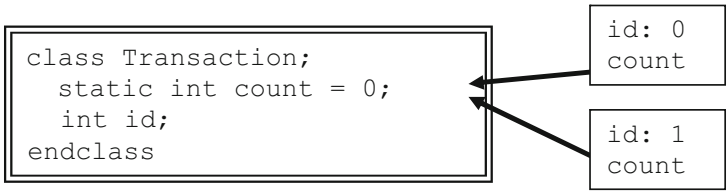

Fig. 5.2 Static variables in a class

Using the ID field is a good way to track objects as they flow through a design. When debugging a testbench, you often need a unique value. SystemVerilog does not let you print the address of an object, but you can make an ID field. Whenever you are tempted to make a global variable, consider making a class-level static variable. A class should be self-contained, with as few outside references as possible.

5.11.2 Accessing Static Variables Through the Class Name

Sample 5.14 showed how you can reference a static variable using a handle. You don't need a handle; you could use the class name followed by `::`, the class scope resolution operator, shown in Sample 5.15.

Sample 5.15 The class scope resolution operator

```
class Transaction;
  static int count = 0;           // Number of objects created
endclass

initial begin
  run_test();
  $display("%0d transactions were created",
           Transaction::count);  // Reference static w/o handle
end
```

5.11.3 Initializing Static Variables

A static variable is usually initialized in the declaration. You can't easily initialize it in the class constructor, as this is called for every single new object. You would need another static variable to act as a flag, indicating whether the original variable had been initialized. If you have a more elaborate initialization, you could use an initial block. Make sure static variables are initialized before the first object is constructed.

Another use for a static variable is when every instance of a class needs information from a single object. For example, a transaction class may refer to a configuration object that has the number of transactions. If you have a non-static handle in the `Transaction` class, every object will have its own copy, wasting space. Sample 5.16 shows how to use a static variable instead.

Sample 5.16 Static storage for a handle

```
class Transaction;
  static Config cfg;      // A handle with static storage
endclass

initial begin
  Transaction::cfg = new(.num_trans(42));
end
```

5.11.4 Static Methods

As you employ more static variables, the code to manipulate them may grow into a full fledged routine. In SystemVerilog you can create a static method inside a class that can read and write static variables, even before the first instance has been created.

Sample 5.17 has a simple static function to display the values of the static variables. SystemVerilog does not allow a static method to read or write non-static variables, such as `id`. You can understand this restriction based on the code below. When the function `display_statics` is called at the end of the example, no `Transaction` objects have been constructed, so no storage has been created for `id` variables.

Sample 5.17 Static method displays static variable

```
class Transaction;
  static Config cfg;
  static int count = 0;
  int id;

  // Static method to display static variables.
  static function void display_statics();
    if (cfg == null)
      $display("ERROR: configuration not set");
    else
      $display("Transaction cfg.num_trans=%0d, count=%0d",
               cfg.num_trans, count);
  endfunction
endclass

Config cfg;
initial begin
  cfg = new(.num_trans(42));       // Pass argument by name
  Transaction::cfg = cfg;
  Transaction::display_statics();  // Static method call
end
```

5.12 Scoping Rules

When writing your testbench, you need to create and refer to many variables. SystemVerilog follows the same basic rules as Verilog, with a few helpful improvements.

A scope is a block of code such as a module, program, task, function, class, or `begin/end` block. The `for` and `foreach` loops automatically create a block so that an index variable can be declared or created local to the scope of the loop.

You can only define new variables in a block. New in SystemVerilog is the ability to declare a variable in an unnamed `begin-end` block.

A name can be relative to the current scope or absolute starting with `$root`. For a relative name, SystemVerilog looks up the list of scopes until it finds a match. If you want to be unambiguous, use `$root` at the start of a name. Variables can not be declared in `$root`, that is, outside of any `module, program` or package.

Sample 5.18 uses the same name in several scopes. Note that in actual code, you would use more meaningful names! The name `limit` is used for a global variable, a program variable, a class variable, a function variable, and a local variable in an initial block. The latter is in an unnamed block, so the label created is tool dependent, along with the signal's hierarchical name.

Sample 5.18 Name scope

```
program automatic top;
  int limit;                  // $root.top.limit

  class Foo;
    int limit, array[];       // $root.top.Foo.limit

    // $root.top.Foo.print.limit
    function void print (input int limit);
      for (int i=0; i<limit; i++)
        $display("%m: array[%0d]=%0d", i, array[i]);
    endfunction
  endclass

  initial begin
    int limit = 3;
    Foo bar;

    bar = new();
    bar.array = new[limit];
    bar.print (limit);
  end
endprogram
```

For testbenches, you can declare variables in the `program` or in the `initial` block. If a variable is only used inside a single `initial` block, such as a counter, you should declare it there to avoid possible name conflicts with other blocks. Note that if you declare a variable in an unnamed block, such as the `initial` in Sample 5.18, there is no hierarchical name that works consistently across all tools.

Declare your classes outside of any `program` or `module` in a `package`. This approach can be shared by all the testbenches, and you can declare temporary variables at the innermost possible level. This style also eliminates a common bug that happens when you forget to declare a variable inside a class. SystemVerilog looks for that variable in higher scopes.

If a block uses an undeclared variable, and another variable with that name happens to be declared in the program block, the class uses it instead, with no warning. In Sample 5.19, the function `Bad::display` did not declare the loop variable `i`, so SystemVerilog uses the program level `i` instead. Calling the function changes the value of `test.i`, probably not what you want!

Sample 5.19 Class uses wrong variable

```
program automatic test;
  int i;  // Program-level variable

  class Bad;
    logic [31:0] data[];

    // Calling this function changes the program variable i
    function void display();
      // Forgot to declare i in next statement
      for (i=0; i<data.size(); i++)
        $display("data[%0d]=%x", i, data[i]);
    endfunction
  endclass
endprogram
```

If you move the class into a package, the class cannot see the program-level variables, and thus won't use them unintentionally as shown in Sample 5.20.

Sample 5.20 Move class into package to find bug

```
package Better;
  class Bad;
    logic [31:0] data[];

    // ** Will not compile because of undeclared i
    function void display();
      for (i = 0; i<data.size(); i++)
        $display("data[%0d]=%x", i, data[i]);
    endfunction
  endclass
endpackage


program automatic test;
  int i;  // Program-level variable
  import Better::*;
  //...
endprogram
```

5.12.1 What is `This`?

When you use a variable name, SystemVerilog looks in the current scope for it, and then in the parent scopes until the variable is found. This is the same algorithm used by Verilog. What if you are deep inside a class and want to unambiguously refer to a class-level object? This style code is most commonly used in constructors,

where the programmer uses the same name for a class variable and an argument. In Sample 5.21, the keyword "`this`" removes the ambiguity to let SystemVerilog know that you are assigning the local variable, `name`, to the class variable, `name`.

Sample 5.21 Using `this` to refer to class variable

```
class Scoping;
  string name;

  function new(input string name);
    this.name = name;     // class name = local name
  endfunction

endclass
```

Some people think this argument naming style makes the code easier to read; others think it is a shortcut by a lazy programmer.

5.13 Using One Class Inside Another

A class can contain an instance of another class, using a handle to an object. This is just like Verilog's concept of instantiating a module inside another module to build up the design hierarchy. A common reason for using containment are code reuse and controlling complexity. For example, every one of your transactions may have a statistics block, including timestamps indicating when the transaction started and ended transmission, and information about all transactions, as shown in Fig. 5.3 and Sample 5.22.

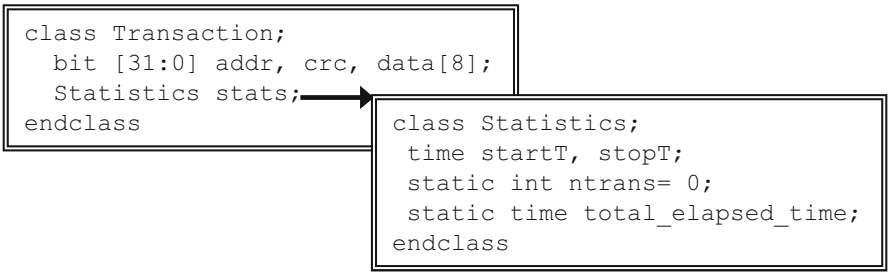

Fig. 5.3 Contained objects Sample 5.22

Sample 5.22 `Statistics` class declaration

```
class Statistics;
  time startT;              // Transaction start time
  static int ntrans = 0;    // Transaction count
  static time total_elapsed_time = 0;

  function void start();
    startT = $time;
  endfunction

  function void stop();
    time how_long = $time - startT;
    ntrans++;               // Another trans completed
    total_elapsed_time += how_long;
  endfunction

endclass
```

Now you can use the Statistics class inside another class such as a transaction as can been seen in Sample 5.23.

Sample 5.23 Encapsulating the `Statistics` class

```
class Transaction;
  bit [31:0] addr, csm, data[8];
  Statistics stats;         // Statistics handle

  function new();
    stats = new();          // Make instance of Statistics
  endfunction

  task transmit_me();
    // Fill packet with data
    stats.start();
    // Transmit packet
     #100;
     stats.stop();
  endtask
endclass
```

The outermost class, `Transaction`, can refer to things in the `Statistics` class using the usual hierarchical syntax, such as `stats.startT`.

Remember to instantiate the object; otherwise, the handle `stats` is `null` and the call to `start` fails. This is best done in the constructor of the outer class, `Transaction`.

As your classes become larger, they may become hard to manage. When your variable declarations and method prototypes grow larger than a page, you should see if there is a logical grouping of items in the class so that it can be split into several smaller ones.

This is also a potential sign that it's time to refactor your code, i.e., split it into several smaller, related classes. See Chapter 8 for more details on class inheritance. Look at what you're trying to do in the class. Is there something you could move into one or more base classes, i.e., decompose a single class into a class hierarchy? A classic indication is similar code appearing at various places in the class. You need to factor that code out into a function in the current class, one of the current class's parent classes, or both.

5.13.1 How Big or Small Should My Class Be?

Just as you may want to split up classes that are too big, you should also have a lower limit on how small a class should be. A class with just one or two members makes the code harder to understand as it adds an extra layer of hierarchy and forces you to constantly jump back and forth between the parent class and all the children to understand what it does. In addition, look at how often it is used. If a small class is only instantiated once, you might want to merge it into the parent class.

One Synopsys customer put each transaction variable into its own class for fine control of randomization. The transaction had a separate object for the address, checksum, data, etc. In the end, this approach only made the class hierarchy more complex. On the next project they flattened the hierarchy.

See Section 8.4 for more ideas on partitioning classes.

5.13.2 Compilation Order Issue

Sometimes you need to compile a class that includes another class that is not yet defined. The declaration of the handle causes an error, as the compiler does not recognize the new type. Declare the class name with a `typedef` statement, as shown in Sample 5.24.

Sample 5.24 Using a `typedef class` statement

```
typedef class Statistics;  // Define a lower level class

class Transaction;
  Statistics stats;        // Use Statistics class
  ...
endclass

class Statistics;          // Define Statistics class
  ...
endclass
```

5.14 Understanding Dynamic Objects

In a statically allocated language such as Verilog, every signal has a unique variable associated with it. For example, there may be a wire called `grant`, the integer `count`, and a module instance `i1`. In OOP, there is not the same one-to-one correspondence. There can be many objects, but only a few named handles. A testbench may allocate a thousand transaction objects during a simulation, but may only have a few handles to manipulate them. This situation takes some getting used to if you have only written Verilog code.

In reality, there is a handle pointing to every active object. Some handles may be stored in arrays or queues, or in another object, like a linked list. For objects stored in a mailbox, the handle is in an internal SystemVerilog structure. See Section 7.6 for more information on mailboxes. Remember that as soon as you assign a new value to the last handle pointing to an object, that object can be garbage collected.

5.14.1 Passing Objects and Handles to Methods

What happens when you pass an object into a method? Perhaps the method only needs to read the values in the object, such as `transmit` above. Or, your method may modify the object, like a method to create a packet. Either way, when you call the method, you pass a handle to the object, not the object itself.

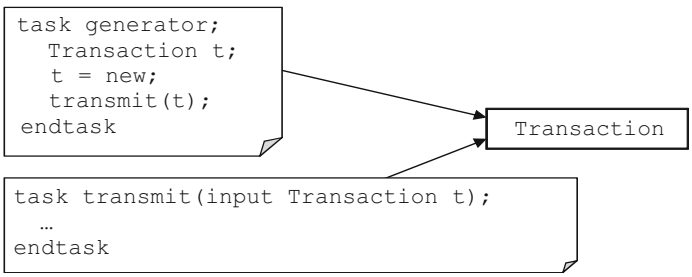

Fig. 5.4 Handles and objects across methods

In Fig. 5.4, the `generator` task has just called `transmit`. There are two handles, `generator.t` and `transmit.t`, that both refer to the same object.

When you call a method, if you pass a scalar variable such as a handle into a `ref` argument, SystemVerilog passes the address of the variable so the method can modify it. If you don't use `ref`, SystemVerilog copies the scalar's value into the argument variable, so any change to the argument in the method does not affect the original value.

Sample 5.25 Passing objects

```
// Transmit a packet onto a 32-bit bus
task transmit(input Transaction tr);
  tr.data[0] = ~tr.data[0];  // Corrupt the first word
  CBbus.rx_data <= tr.data[0];
  ...
endtask

Transaction tr;
initial begin
  tr = new();        // Allocate the object
  tr.addr = 42;      // Initialize values
  transmit(tr);      // Pass object to task
end
```

In Sample 5.25, the initial block allocates a `Transaction` object and calls the `transmit` task with the handle that points to the object. Using this handle, `transmit` can read and write values in the object. However, if `transmit` tries to modify the handle, the result won't be seen in the initial block, as the `t` argument was not declared as `ref`.

A method can modify an object, even if the handle argument does not have a `ref` modifier. This frequently causes confusion for new users, as they mix up the handle with the object. As shown above, `transmit` can modify `data[0]` in the object without changing the value of `t`. If you don't want an object modified in a method, pass a copy of it so that the original object is untouched. See Section 5.15 for more on copying objects.

5.14.2 Modifying a Handle in a Task

A common coding mistake is to forget to use `ref` on method arguments that you want to modify, especially handles. In Sample 5.26, the argument `tr` is not declared as `ref`, so any change to it is not be seen by the calling code. The argument `tr` has the default direction of `input`.

Sample 5.26 Bad transaction creator task, missing `ref` on handle

```
function void create(Transaction tr); // Bug, missing ref
  tr = new();
  tr.addr = 42;
  // Initialize other fields
  ...
endfunction

Transaction t;
initial begin
  create(t);             // Create a transaction
  $display(t.addr);      // Fails because t=null
end
```

Even though `create` modified the argument `tr`, the handle `t` in the calling block remains `null`. You need to declare the argument `tr` as `ref` as can be seen in Sample 5.27.

Sample 5.27 Good transaction creator task with ref on handle

```
function void create(ref Transaction tr);
  ...
endfunction : create
```

If a method is only going to modify the properties of the object, the method should declare the handle as an input argument. If a method is going to modify the handle, for example to make it point to a new object, the method must declare the handle as a ref argument.

5.14.3 Modifying Objects in Flight

A very common mistake is forgetting to create a new object for each transaction in the testbench. In Sample 5.28, the `generate_bad` task creates a `Transaction` object with random values, and transmits it into the design over several cycles.

Sample 5.28 Bad generator creates only one object

```
task generator_bad(input int n);
  Transaction t;
  t = new();                    // Create one new object
  repeat (n) begin
    t.addr = $random();         // Initialize variables
    $display("Sending addr=%h", t.addr);
    transmit(t);                // Send it into the DUT
  end
endtask
```

What are the symptoms of this mistake? The code above creates only one `Transaction`, so every time through the loop, `generator_bad` changes the object at the same time it is being transmitted. When you run this, the `$display` shows many `addr` values, but all transmitted `Transaction` objects have the same value of `addr`. The bug becomes visible when `transmit` spawns off a thread that takes several cycles to send the transaction, and so the values in the object are re-randomized in the middle of transmission. If your `transmit` task makes a copy of the object, you can recycle the same object over and over. This bug can also happen with mailboxes as shown in Sample 7.32

To avoid this bug, you need to create a new `Transaction` during each pass through the loop as seen in Sample 5.29.

Sample 5.29 Good generator creates many objects

```
task generator_good(input int n);
  Transaction t;
  repeat (n) begin
    t = new();                  // Create one new object
    t.addr = $random();         // Initialize variables
    $display("Sending addr=%h", t.addr);
    transmit(t);                // Send it into the DUT
  end
endtask
```

5.14.4 Arrays of Handles

As you write testbenches, you need to be able to store and reference many objects. You can make arrays of handles, each of which refers to an object. Sample 5.30 shows storing ten bus transaction handles in an array.

Sample 5.30 Using an array of handles

```
task generator();
  Transaction tarray[10];
  foreach (tarray[i]) begin
    tarray[i] = new();       // Construct each object
    transmit(tarray[i]);
  end
endtask
```

The array `tarray` is made of handles, not objects. So you need to construct each object in the array before using it, just as you would for a normal handle. There is no way to call **new** on an entire array of handles.

There is no such thing as an "array of objects", though you may use this term as a shorthand for an array of handles that points to objects. Keep in mind that some handles may be set to `null`, or that multiple handles could point to a single object.

5.15 Copying Objects

You may want to make a copy of an object to keep a method from modifying the original, or in a generator to preserve the constraints. You can either use the simple, built-in copy available with `new` operator or you can write your own for more complex classes. See Section 8.2 for more reasons why you should make a copy method.

5.15.1 Copying an Object with the `New` Operator

Copying an object with the `new` operator is easy and reliable as shown in Sample 5.31. Memory for the new object is allocated and all variables from the existing object are copied. However any `new()` function that you may have defined is not called.

Sample 5.31 Copying a simple class with `new`

```
class Transaction;
  bit [31:0] addr, csm, data[8];
  function new();
    $display("In %m");
  endfunction
endclass

Transaction src, dst;
initial begin
  src = new();         // Create first object
  dst = new src;       // Make a copy with new operator
end
```

This is a shallow copy, similar to a photocopy of the original, blindly transcribing values from source to destination. If the class contains a handle to another class, only the handle's value is copied by the `new` operator, not a full copy of the lower level object. In Sample 5.32, the `Transaction` class contains a handle to the `Statistics` class, originally shown in Sample 5.22.

Sample 5.32 Copying a complex class with `new operator`

```
class Transaction;
  bit [31:0] addr, csm, data[8];
  static int count = 0;
  int id;
  Statistics stats;        // Handle points to Statistics object

  function new();
    stats = new();          // Construct a new Statistics object
    id = count++;
  endfunction
endclass

Transaction src, dst;
initial begin
  src = new();            // Create a Transaction object
  src.stats.startT = 42; // Results in Figure 5-5
  dst = new src;          // Copy src to dst with new operator
                          // Results in Figure 5-6
  dst.stats.startT = 96; // Changes stats for dst & src
  $display(src.stats.startT); // 96, see Figure 5-7
end
```

The initial block creates the first **`Transaction`** object and modifies a variable in the contained object **`stats`** as shown in Fig. 5.5.

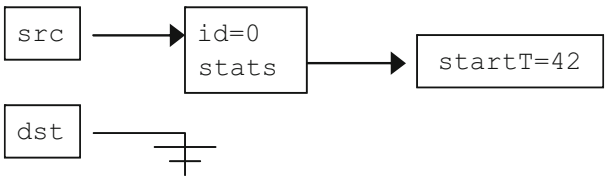

Fig. 5.5 Objects and handles before copy with the `new` operator

When you use the `new` operator to make a copy, the **`Transaction`** object is copied, but not the **`Statistics`** one. This is because the `new` operator does not call your own `new()` function. Instead, the values of variables and handles are copied. So now both **`Transaction`** objects have the same `id` as shown in Fig. 5.6.

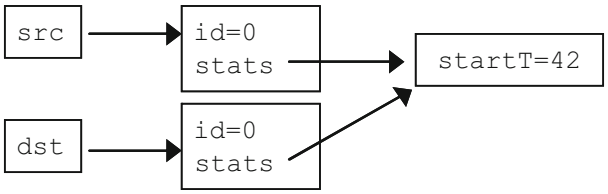

Fig. 5.6 Objects and handles after copy with the `new` operator

Worse yet, both **`Transaction`** objects point to the same **`Statistics`** object so modifying `startT` with the `src` handle affects what is seen with the `dst` handle as you can see in Figure 5.7.

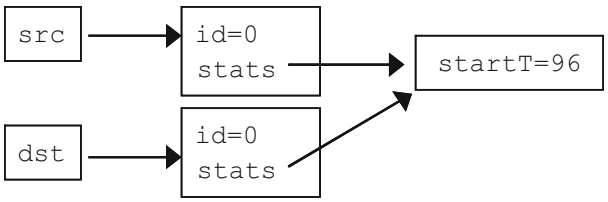

Fig. 5.7 Both src and dst objects refer to a single statistics object and see updated startT value

5.15.2 Writing Your Own Simple Copy Function

If you have a simple class that does not contain any references to other classes, writing a `copy` function is easy as you can see in Samples 5.33 and 5.34. Instead of calling the `new()` function and copying each individual variable, the `copy` function could have instead used the `new` operator, but then it would need to replicate any processing done in `new()`, such as setting the `id`.

Sample 5.33 Simple class with `copy` function

```
class Transaction;
  bit [31:0] addr, csm, data[8];  // No Statistic handle

  function Transaction copy();
    copy = new();                 // Construct destination
    copy.addr = addr;             // Fill in data values
    copy.csm  = csm;
    copy.data = data;             // Array copy
  endfunction
endclass
```

Sample 5.34 Using a `copy` function

```
Transaction src, dst;
initial begin
  src = new();                    // Create first object
  dst = src.copy();               // Make a copy of the object
end
```

5.15.3 *Writing a Deep Copy Function*

For nontrivial classes, you should always create your own `copy` function as seen in Sample 5.35. You can make it a deep copy by calling the `copy` functions of all the contained objects. Your own `copy` function makes sure all your user fields (such as `id`) remain consistent. The downside of making your own `copy` function is that you need to keep it up to date as you add new variables – forget one and you could spend hours debugging to find the missing value.

Sample 5.35 Complex class with deep copy function

```
class Transaction;
  bit [31:0] addr, csm, data[8];
  Statistics stats;         // Handle points to Statistics object
  static int count = 0;
  int id;

  function new();
    stats = new();
    id = count++;
  endfunction

  function Transaction copy();
    copy = new();               // Construct destination object
    copy.addr = addr;           // Fill in data values
    copy.csm  = csm;
    copy.data = data;
    copy.stats = stats.copy(); // Call Statistics::copy
  endfunction
endclass
```

The `new()` constructor is called by `copy` so every object gets a unique `id`. Add a `copy()` method for the `Statistics` class as shown in Sample 5.36, and every other class in the hierarchy.

Sample 5.36 `Statistics` class declaration

```
class Statistics;
  time startT;       // Transaction times
  ...                // See Sample 5-22 for rest of class
  function Statistics copy();
    copy = new();
    copy.startT = startT;
  endfunction
endclass
```

Now when you make a copy of the `Transaction` object, it will have its own `Statistics` object as shown in Sample 5.37.

Sample 5.37 Copying a complex class with `new operator`

```
Transaction src, dst;
initial begin
  src = new();                 // Create first object
  src.stats.startT = 42;       // Set start time
  dst = src.copy();            // Deep copy src to dst
  dst.stats.startT = 96;       // Changes stats for dst only
  $display(src.stats.startT); // "42", See Figure 5-8
end
```

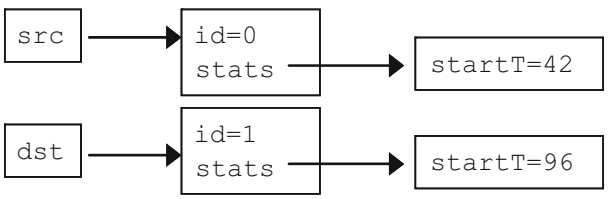

Fig. 5.8 Objects and handles after deep copy

The good news is that the UVM data macros create the `copy` function automatically, so you are spared from having to write them by hand. Manually creating these is very error prone, especially when you add new variables.

5.15.4 *Packing Objects to and from Arrays Using Streaming Operators*

Some protocols, such as ATM, transmit control and data values one byte at a time. Before you send out a transaction, you need to pack together the variables in the object to a byte array. Likewise, after receiving a string of bytes, you need to unpack them back into a transaction object. For both of these functions, use the streaming operators, as shown in Section 2.12.

You can't just stream the entire object as this would gather all properties, including both data and also meta-data such as timestamps and self-checking information that you may not want packed. You need to write your own `pack` function like the one in Samples 5.38 and 5.39 that only uses the properties that you choose.

More good news - the UVM data macros create the pack and unpack methods.

Sample 5.38 Transaction class with pack and unpack functions

```
class Transaction;
  bit [31:0] addr, csm, data[8]; // Real data
  static int count = 0;          // Meta-data does not
  int id;                        //     get packed

  function new();
    id = count++;
  endfunction

  function void display();
    $write("Tr: id=%0d, addr=%x, csm=%x", id, addr, csm);
    foreach(data[i]) $write(" %x", data[i]);
    $display;
  endfunction

  function void pack(ref byte bytes[$]);
    bytes = { >> {addr, csm, data}};
  endfunction

  function Transaction unpack(ref byte bytes[$]);
    { >> {addr, csm, data}} = bytes;
  endfunction
endclass : Transaction
```

Sample 5.39 Using the pack and unpack functions

```
Transaction tr, tr2;
byte b[$];                  // Queue of bytes

initial begin
  tr = new();
  tr.addr = 32'ha0a0a0a0;   // Fill object with values
  tr.csm =  '1;
  foreach (tr.data[i])
    tr.data[i] = i;

  tr.pack(b);               // Pack object into byte array
  $write("Pack results: ");
  foreach (b[i])
    $write("%h", b[i]);
   $display;

  tr2 = new();
  tr2.unpack(b);
  tr2.display();
end
```

5.16 Public vs. Local

The core concept of OOP is encapsulating data and related methods into a class. Variables are kept local to the class by default to keep one class from poking around inside another. A class provides a set of accessor methods to access and modify the data. This would also allow you to change the implementation without needing to let the users of the class know. For instance, a graphics package could change its internal representation from Cartesian coordinates to polar as long as the user interface (accessor methods) have the same functionality.

Consider the `Transaction` class that has a payload and a checksum so that the hardware can detect errors. In conventional OOP, you would make a method to set the payload also set the checksum so they would stay synchronized. Thus your objects would always be filled with correct values.

However, testbenches are not like other programs, such as a web browser or word processor. A testbench needs to create errors. You want to have a bad checksum so you can test how the hardware reacts to errors.

OOP languages such as C++ and Java allow you to specify the visibility of variables and methods. By default, everything in a class is local unless labeled otherwise.

In SystemVerilog, everything is public unless labeled `local` or `protected`. You should stick with this default so you have the greatest control over the operation of the DUT, which is more important than long-term software stability. For example, making the checksum visible allows you to easily inject errors into the

DUT. If the checksum was local, you would have to write extra code to bypass the data-hiding mechanisms, resulting in a larger and more complex testbench.

5.17 Straying Off Course

As a new OOP student, you may be tempted to skip the extra thought needed to group items into a class, and just store data in a few variables. Avoid the temptation! A basic DUT monitor samples several values from an interface. Don't just store them in some integers and pass them to the next stage. This saves you a few minutes at first, but eventually you need to group these values together to form a complete transaction. Several of these transactions may need to be grouped to create a higher-level transaction such as a DMA transfer. Instead, immediately put those interface values into a transaction class. Now you can store related information (port number, receive time) along with the data, and easily pass this object to the rest of your testbench.

5.18 Building a Testbench

Now that you have seen the basics of OOP, you can see how to create a layered testbench from a set of classes. Figure 5.9 is the diagram from Chapter 1. Obviously, the transactions flowing between the blocks are objects, but each block is also modeled with a class.

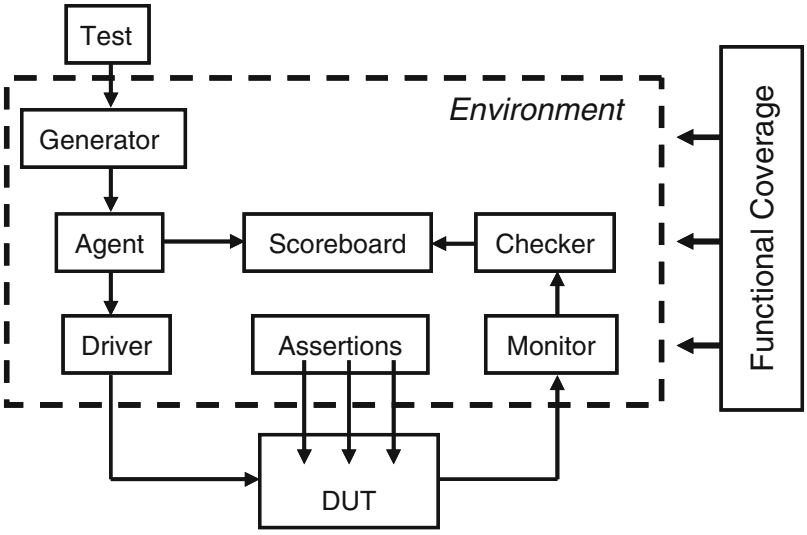

Fig. 5.9 Layered testbench

The `Generator`, `Agent`, `Driver`, `Monitor`, `Checker`, and `Scoreboard` are all classes, modeled as transactors (described below). They are instantiated inside the Environment class. For simplicity, the test is at the top of the hierarchy, as is the program that instantiates the Environment class. The Functional coverage definitions can be put inside or outside the Environment class. See Section 1.10 for a description of the layered verification environment and its components.

A transactor is made of a simple loop that receives a transaction object from a previous block, makes some transformations, and sends it to the following one as you can see in Sample 5.40. Some, such as the `Generator`, have no upstream block, so this transactor constructs and randomizes every transaction, while others, such as the `Driver`, receive a transaction and send it into the DUT as signal transitions.

Sample 5.40 Basic Transactor

```
class Transactor;  // Generic class
  Transaction tr;

  task run();
    forever begin
      // Get transaction from upstream block
      ...
      // Do some processing
      ...
      // Send it to downstream block
      ...
    end
  endtask

endclass
```

How do you exchange transactions between blocks? With procedural code you could have one object call the next, or you could use a data structure such as a FIFO to hold transactions in flight between blocks. In Chapter 7, you will learn how to use mailboxes, which are FIFOs with the ability to stall a thread until a new value is added.

5.19 Conclusion

Using Object-Oriented Programming is a big step, especially if your first computer language was Verilog. The payoff is that your testbenches are more modular and thus easier to develop, debug, and reuse.

Have patience — your first OOP testbench may look more like Verilog with a few classes added. As you get the hang of this new way of thinking, you begin to create and manipulate classes for both transactions and the transactors in the testbench that manipulate them.

In Chapter 8 you will learn more OOP techniques so your test can change the behavior of the underlying testbench without having to change any of the existing code.

5.20 Exercises

1. Create a class called `MemTrans` that contains the following members, then construct a `MemTrans` object in an `initial` block.
   a. An 8-bit `data_in` of logic type
   b. A 4-bit `address` of logic type
   c. A void function called `print` that prints out the value of `data_in` and `address`
2. Using the `MemTrans` class from Exercise 1, create a custom constructor, the `new` function, so that `data_in` and `address` are both initialized to 0.
3. Using the `MemTrans` class from Exercise 1, create a custom constructor so that `data_in` and `address` are both initialized to 0 but can also be initialized through arguments passed into the constructor. In addition, write a program to perform the following tasks.
   a. Create two new `MemTrans` objects.
   b. Initialize `address` to 2 in the first object, passing arguments by name.
   c. Initialize `data_in` to 3 and `address` to 4 in the second object, passing arguments by name.
4. Modify the solution from Exercise 3 to perform the following tasks.
   a. After construction, set the `address` of the first object to 4'hF.
   b. Use the `print` function to print out the values of `data_in` and `address` for the two objects.
   c. Explicitly deallocate the 2nd object.
5. Using the solution from Exercise 4, create a static variable `last_address` that holds the initial value of the `address` variable from the most recently created object, as set in the constructor. After allocating objects of class `MemTrans` (done in Exercise 4) print out the current value of `last_address`.
6. Using the solution from Exercise 5, create a static method called `print_last_address` that prints out the value of the static variable `last_address`. After allocating objects of class `MemTrans`, call the method `print_last_address` to print out the value of `last_address`.

7. Given the following code, complete the function `print_all` in class `MemTrans` to print out `data_in` and `address` using the class `PrintUtilities`. Demonstrate using the function `print_all`.

```
class PrintUtilities;

  function void print_4(input string name,
                        input [3:0] val_4bits);
    $display("%t: %s = %h", $time, name, val_4bits);
  endfunction

  function void print_8(input string name,
                        input [7:0] val_8bits);
    $display("%t: %s = %h", $time, name, val_8bits);
  endfunction

endclass


class MemTrans;
  bit [7:0] data_in;
  bit [3:0] address;
  PrintUtilities print;

  function new();
    print = new();
  endfunction

  function void print_all;
    // Fill in function body
  endfunction

endclass
```

8. Complete the following code where indicated by the comments starting with //.

```
program automatic test;
  import my_package::*;  // Define class Transaction

  initial begin
    // Declare an array of 5 Transaction handles
    // Call a generator task to create the objects
  end

  task generator(…);     // Complete the task header
    // Create objects for every handle in the array
    // and transmit the object.
  endtask

  task transmit(Transaction tr);
    .......
  endtask : transmit

endprogram
```

9. For the following class, create a `copy` function and demonstrate its use. Assume the `Statistics` class has its own `copy` function.

```
package automatic my_package;
  class MemTrans;
    bit [7:0] data_in;
    bit [3:0] address;
    Statistics stats;
    function new();
      data_in = 3;
      address = 5;
      stats = new();
    endfunction
  endclass;
endpackage
```