2

Fundamentals of Object-Oriented Programming

Software engineering, unconstrained by the physics of electricity and magnetism, has long sought to build reusable, interchangeable, robust components. An important programming model that addresses the problem is called *object-oriented programming* (OOP). The central idea of OOP is that programs are organized as a collection of interacting objects, each with its own data space and functions. Objects can be made reusable because they encapsulate everything they need to operate, can be built with minimal or no external dependencies, and can be highly parameterized.

This chapter introduces the basic concepts of OOP, including the notions of encapsulation and interface. The chapter concludes with a discussion of why OOP is important for building testbenches.

2.1   Procedural vs. OOP

To understand OOP and the role it plays in verification, it is beneficial to first understand traditional procedural programming and its limitations. This sets the foundation for understanding how OOP can overcome those limitations.

In the early days of assembly language programing, programmers and computer architects quickly discovered that programs often contained sequences of instructions that were repeated throughout a program. Repeating lots of code (particularly with a card punch) is tedious and error prone. Making a change to the sequence involved locating each place the sequence appeared in the program and repeating the change in each location. To avoid the tedium and the errors caused by repeated sequences, the subroutine was invented.

A subroutine is a unit of reusable code. Instead of coding the same sequence of instructions inline, you call a subroutine. Parameters passed to subroutines allow you to dynamically modify the code. That is, each call to a subroutine with different values for the parameters causes the subroutine to behave differently based on the specific parameter values.

Every programming language of any significance has constructs for creating subroutines, procedures, or functions, along with syntax for passing in parameters and returning values. These features are useful for creating operations that are used often. However, some operations are very common (such as I/O, data conversions, numerical methods, and so forth). And to avoid having to rewrite these operations repeatedly, programmers found it valuable to create libraries of commonly used functions. As a result, most programming languages include such a library as part of the compiler package. One of the most well-known examples is the C library that comes with every C compiler. It contains useful functions such as `printf()`, `cos()`, `atof()`, and `qsort()`. These are functions that virtually every programmer will use at some time or another.

Imagine having to write your own I/O routines or your own computation for converting numbers to strings and strings to numbers. There was a time when programmers did just that. Libraries of reusable functions changed all that and increased overall programming productivity.

As software practice and technology advanced, programmers began thinking at higher levels of abstraction than instructions and subroutines. Instead of writing individual instructions, programmers now code in languages that provide highly abstracted models of the computer, and compilers or interpreters translate these models into specific instructions. A library, such as the C library or STL in C++, is a form of abstraction. It presents a set of functions that programmers can use to construct ever more complex programs or abstractions.

In his seminal book *Algorithms + Data Structures = Programs*, Niklaus Wirth explains that to solve any programming problem, you must devise an abstraction of reality that has the characteristics and properties of the problem at hand and ignore the rest of the details. He argues that the collection of data you need to solve a problem forms the abstraction. So before you can solve a problem, you first need to determine what data you need to have to create the solution.

To continue building reusable abstractions, we need to create libraries of data objects that can be reused to solve specific kinds of problems. The search for ways to do this leads to the development of object-oriented technology.

Object-oriented program analysis and design is centered around data objects, the functionality associated with each object, and the relationships between objects.

The goal of OOP is to facilitate *separation of concerns*, a phrase coined by Edsger Dijkstra in his 1974 essay titled, "On the Role of Scientific Thought."[1] In this essay he quotes himself:

> It is what I sometimes have called "the separation of concerns," which, even if not perfectly possible, is yet the only available technique for effective ordering of one's thoughts, that I know of. This is what I mean by "focussing one's attention upon some aspect": it does not mean ignoring the other aspects, it is just doing justice to the fact that from this aspect's point of view, the other is irrelevant. It is being one- and multiple-track minded simultaneously....

Object-oriented languages provide facilities to separate program concerns and focus on them independently, and, to encapsulate data abstractions and present them through well-defined interfaces. Complete object-oriented programs are constructed by separating the program's functionality into distinct classes, defining the interfaces for each class, and then establishing connections and interactions between components through their interfaces.

2.2   Classes and Objects

The primary unit of programming in object-oriented languages, such as SystemVerilog, is the *class*. A class contains data elements, called *members*, and tasks and functions, called *methods*. To execute an object-oriented program, you must *instantiate* one or more classes in a main routine and then call methods on the various objects. Although the terms class and object are sometimes used interchangeably, typically, the term *class* refers to a class declaration or an uninstantiated object, and the term *object* refers to an instance of a class.

To illustrate these concepts, below is an example of a simple class called `register`.

```
class register;
  local bit[31:0] contents;

  function void write(bit[31:0] d)
    contents = d;
```

---

1. The complete text of Dijkstra's essay is at http://www.cs.utexas.edu/users/EWD/ewd04xx/EWD447.PDF

```
    endfunction

    function bit[31:0] read();
       return contents;
    endfunction
endclass
```

This very simple class has one member, `contents`, and two methods, `read()` and `write()`. To use this class, you create objects by instantiating the class and then call the object's methods, as shown below

```
module top;
   register r;
   bit[31:0] d;

   initial begin
      r = new();
      r.write(32'h00ff72a8);
      d = r.read();
   end
endmodule
```

The `local` attribute on class member `contents` tells the compiler to strictly enforce the boundaries of the class. If you try to access `contents` directly, the compiler issues an error. You can only access contents through the publicly available read and write functions. This kind of access control is important to guarantee no dependencies on the internals of the class and thus enable the class to be reused.

You can use classes to create new data types, such as our simple `register`. Using classes to create new data types is an important part of OOP. You can also use them to encapsulate mathematical computations or to create dynamic data structures, such as stacks, lists, queues, and so forth. Encapsulating the organization of a data structure or the particulars of a computation in a class makes the data structure or computation highly reusable.

As a more complete example, let's look at a useful data type, the pushdown stack. A stack is a LIFO (last in first out) structure. Items are put into the stack with `push()`, and items are retrieved from the stack with `pop()`. `pop()` returns the last item pushed and removes it from the data structure. The internal member `stkptr` keeps track of the top of the stack. The item it points to is the top, and everything below it (that is, with a smaller index) is lower in the stack. Below is a basic implementation of a stack in SystemVerilog.

```
43    class stack;
44
```

```
45        typedef bit[31:0] data_t;
46        local data_t stk[20];
47        local int stkptr;
48
49        function new();
50          clear();
51        endfunction
52
53        function bit pop(output data_t data);
54
55          if(is_empty())
56            return 0;
57
58          data = stk[stkptr];
59          stkptr = stkptr - 1;
60          return 1;
61
62        endfunction
63
64        function bit push(data_t data);
65
66          if(is_full())
67            return 0;
68
69          stkptr = stkptr + 1;
70          stk[stkptr] = data;
71          return 1;
72
73        endfunction
74
75        function bit is_full();
76          return stkptr >= 19;
77        endfunction
78
79        function bit is_empty();
80          return stkptr < 0;
81        endfunction
82
83        function void clear();
84          stkptr = -1;
85        endfunction
86
87        function void dump();
88
89          $write("stack:");
90          if(is_empty()) begin
91            $display("<empty>");
92            return;
93          end
94
95          for(int i = 0; i <= stkptr; i = i + 1) begin
96            $write(" %0d", stk[i]);
97          end
```

```
98
99        if(is_full())
100         $write(" <full>");
101        $display("");
102
103    endfunction
104  endclass
file: 02_intro_to_OOP/01_stack/stack.sv
```

The class `stack` encapsulates everything there is to know about the stack data structure. It contains an *interface* and an *implementation* of the interface. The interface is the set of methods that you use to interact with the class. The implementation is the behind-the-scenes code that makes the class operate. The interface to our stack contains the following methods:

```
function new();
function bit pop(output DATA data);
function bit push(DATA data);
function bit is_full();
function bit is_empty();
function void clear();
function void dump();
```

There is no other way to interact with `stack` than through these methods. There are also two data members of the class, `stk` and `stkptr`, that represent the actual stack structure. However, these two members are local, which means that the compiler will disallow any attempts to access them from outside the class. By preventing access to the internals of the data structure from outside, we can make some guarantees about the state of the data. For example, `push()` and `pop()` can rely on the fact that `stkptr` is correct and points to the top of the stack. If it were possible to change the value of `stkptr` by means other than using the interface functions, then `push()` and `pop()` would have to resort to additional time-consuming and possibly unreliable checks to determine the validity of `stkptr`.

The implementation of the interface occurs inline. The class declaration contains not only the interface definition, but also the implementation of each of the interface functions. Both C++ and SystemVerilog allow the implementation to be separate from the interface. Separating the interface and the implementation is an important concept. Programmers writing in C++ can use header files to capture the interface and `.cc` (or `.cpp` or whatever the compiler uses) to hold the implementation.

There are some important by-products of enforcing access through class interfaces. One is reusability. We can more easily reuse classes whose

interfaces are well defined and well explained than those whose interfaces are fuzzy. Another important by-product of enforcing access through class interfaces is reliability. The authors of the class can guarantee certain invariants (for example, `stkptr` is less than the size of the available `stk` array) when they know that users will not modify the data other than by the means provided. In addition, users can expect the state of the object to be predictable when they adhere to the interface. Clarity is another by-product. An interface can describe the entire semantics of the class. The object will do nothing other than execute the operations available through the interface. This makes it easier for those who use the class to understand exactly what it will do.

2.3   Object Relationships

The true power of OOP becomes apparent when objects are connected in various relationships. There are many kinds of relationships that are possible. We will consider two of the most fundamental relationships HAS-A and IS-A.

2.3.1   HAS-A

HAS-A refers to the concept of one object contained or owned by another. The HAS-A relationship is represented by members. In our stack class, for example, the stack HAS-A stack pointer (`stkptr`) and stack array. Those are primitive data types, not classes, but the same concept of HAS-A applies. In SystemVerilog you can create HAS-A relationships between classes with references or pointers. The figure below illustrates the underlying memory model for a HAS-A relationship. Object A contains a reference or a pointer to object B.
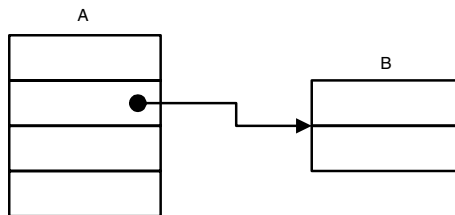

Figure 2-1 HAS-A Relationship

The Unified Modeling Language (UML) is a graphical language for representing systems, particularly the relationships between objects in those

systems. The UML for a HAS-A relationship is expressed with a line between objects and a filled-diamond arrowhead, as in the diagram below.

Figure 2-2 UML for a HAS-A Relationship

Object A owns an instance of object B. Coding a HAS-A relationship in SystemVerilog involves instantiating one class inside another or in some other way providing a handle to one class that is stored inside another.

```
class B;
endclass

class A;
   local B b;
   function new();
      b = new();
   endfunction
endclass
```

class A contains a reference to class B. The constructor for class A, function new(), calls new() on class B to create an instance of it. The member b holds a reference to the newly created instance of B.

2.3.2 IS-A

The IS-A relationship is most often referred to as *inheritance*. A new class is *derived* from a previously existing object and *inherits* its characteristics. Objects created with inheritance are composed using IS-A. The derived object is considered a sub-class or a more specialized version of the parent object.

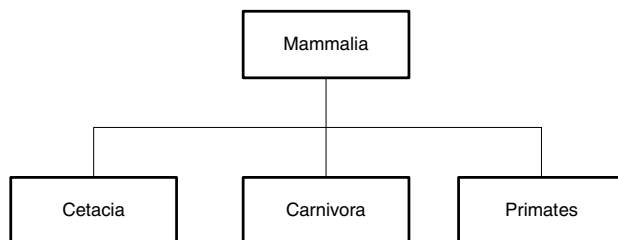To illustrate the notion of inheritance, Figure 2-3 uses a portion of the taxonomy of mammals.

Figure 2-3 IS-A Example: Mammal Taxonomy

Animals that are members of the cetacia, carnivora, or primate orders are mammals. These very different kinds of creatures share the common traits of mammals. Yet cetacia (whales, dolphins), carnivora (dogs, bears, raccoons), and primates (monkeys, humans) each have their distinct and unmistakable characteristics. To use OO terminology, a bear IS-A carnivore and a carnivore IS-A mammal. In other words, a bear is composed of attributes of both mammals and carnivores plus additional attributes that distinguish it from other carnivores.

To express IS-A using UML, we draw a line between objects with an open arrow head pointing to the base class. Traditionally, we draw the base class above the derived classes, and the arrows point upward, forming an inheritance tree (or a directed acyclic graph that can be implemented in languages, such as C++, that support multiple inheritance).
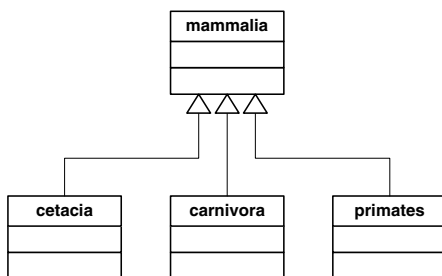

Figure 2-4 UML for IS-A Relationship

When composing two objects together in a computer program using inheritance, the new derived object contains characteristics of the parents and

usually includes additional characteristics. The figure below illustrates the underlying memory model for an IS-A composition. In the example, the class B is *derived* from A.
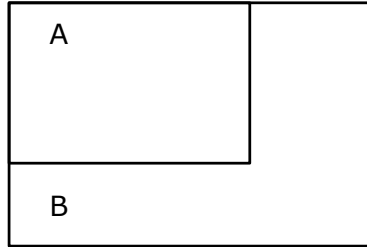

Figure 2-5 Example of IS-A Relationship

SystemVerilog uses the keyword `extends` to identify an inheritance relationship between classes:

```
class A;
   int i;
   float f;
endclass

class B extends A;
   string s;
endclass
```

Class B is derived from A, so it contains all the attributes of A. Any instance of B not only contains the string s, but also the floating point value f and the integer i.

2.4   Virtual Functions and Polymorphism

One of the reasons for composing objects through inheritance is to establish different behaviors for the same operation. In other words, the behavior defined in a derived class overrides behavior defined in a base class. The means to do this is through *virtual functions*. A virtual function is one that can be overridden in a derived class. Consider the following generic packet class.

```
class generic_packet;
   addr_t src_addr;
   addr_t dest_addr;
   bit m_header [];
   bit m_trailer []'
   bit m_body [];

   virtual function void set_header();
```

```
      virtual function void set_trailer();
      virtual function void set_body();
    endclass
```

It has three virtual functions to set the contents of the packet. Different kinds of packets require different kinds of contents. We use `generic_packet` as a base class and derive different kinds of packets from it.

```
    class packet_A extends generic packet;
      virtual function void set_header();
      endfunction
      virtual function void set_trailer();
      endfunction
      virtual function void set_body();
      endfunction
    endclass

    class packet_B extends generic_packet;
      virtual function void set_header();
      endfunction
      virtual function void set_trailer();
      endfunction
      virtual function void set_body();
      endfunction
    endclass
```

Both `packet_A` and `packet_B` may have different headers and trailers and different payload formats. The knowledge about how the parts of the packet are formatted is kept locally inside the derived packet classes. The virtual functions `set_header()`, `set_trailer()`, and `set_body()` are implemented differently in each subclass based on the packet type. The base class `generic_packet` establishes the organization of the class and the types of operations that are possible, and the derived classes can modify the behavior of those operations.

Virtual functions are used to support *polymorphism*: multiple classes that can be used interchangeably, each with different behaviors. For example, some processing of packets may not need to know what kind of packet is being processed. The only information necessary is that the object is indeed a packet; that is, it is derived from the base class. Another way to say that is, the the current packet is related to the base class packet via the IS-A relationship. Virtual functions are the mechanism by which we can code alternate behaviors for different variations of a packet.

To look a little deeper at how virtual functions work, let's consider three classes related to each other by the IS-A relationship.
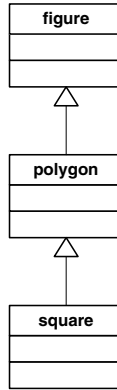

Figure 2-6 Three Classes Related with IS-A

`figure` is the base class; `polygon` is derived from `figure`; `square` is derived from `polygon`. Each class has two functions, `draw()`, which is virtual, and `compute_area()`, which is non-virtual. The following sample shows the SystemVerilog code:

```
38
39    class figure;
40
41       virtual function void draw();
42          $display("figure::draw");
43       endfunction
44
45       function void compute_area();
46          $display("figure::compute_area");
47       endfunction
48
49    endclass
50
51    class polygon extends figure;
52
53       virtual function void draw();
54          $display("polygon::draw");
55       endfunction
56
57       function void compute_area();
58          $display("polygon::compute_area");
59       endfunction
60
61    endclass
```

```
62
63    class square extends polygon;
64
65      virtual function void draw();
66         $display("square::draw");
67      endfunction
68
69      function void compute_area();
70         $display("square::compute_area");
71      endfunction
72
73    endclass
file: 02_intro_to_OOP/03_virtual/virtual.sv
```

Each function prints out its fully qualified name in the form
class_name::function_name. We can write a simple program that calls each
of these functions to understand how the virtual functions are bound.

```
75    program top;
76       figure f;
77       polygon p;
78       square s;
79
80       initial begin
81          s = new();
82          f = s;
83          p = s;
84
85          p.draw();
86          p.compute_area();
87          f.draw();
88          f.compute_area();
89          s.draw();
90          s.compute_area();
91       end
92    endprogram
file: 02_intro_to_OOP/03_virtual/virtual.sv
```

The following shows what happens when we run this program:

```
square::draw
polygon::compute_area
square::draw
figure::compute_area
square::draw
square::compute_area
```

First we create s, a square, and then we assign it to f and p. The immediate
base class of square is polygon and the base class of polygon is figure. From

the printed output, we can conclude that the functions are bound according to the following table:

`p.draw()`	`square::draw()`
`p.compute_area()`	`polygon::compute_area()`
`f.draw()`	`square::draw()`
`f.compute_area()`	`figure::compute_area()`
`s.draw()`	`square::draw()`
`s.compute_area()`	`square::compute_area()`

In all cases, `compute_area()` was bound to the particular `compute_area()` function specified by the type of the reference that called it—`p` is a reference to a `polygon`, thus `polygon::compute_area()` is bound. This is because `compute_area()` is non-virtual. The compiler can easily determine which version of the function to call simply based on the type of the object.

Because `draw()` is virtual, it is not always possible for the compiler to determine which function to call. The decision is made at run time using a *virtual table*, a table of function bindings. A virtual table is used to bind functions whose bindings cannot be entirely determined at compile time. A good reference for learning more about how virtual tables work is *Inside the C++ Object Model* by Stanley B. Lippman.

Notice that even though `p` is a `polygon`, the call to `p.draw()` results in `square::draw()` being called not `polygon::draw()`, as you might expect. The same thing happens with `f`—`f.draw()` is bound to `square::draw()`. The object we originally instantiated is a `square`, and even though we assign handles of different types, the fact that it is a `square` is not forgotten. This works only because `square` is derived from `polygon`, which in turn is derived from `figure`, and because `draw()` is declared as virtual. A compile time error about type incompatibility occurs if you try to assign `s` to `p` and `s` is not derived from `p`.

2.5   Generic Programming

Recall that object-oriented languages provide facilities to separate program concerns and focus on them independently. An implication of separating concerns is that each concern is represented only once. Duplicating code violates the principle. In practice, many problems are quite similar, and their solution requires code that is similar, but not identical. Intuitively, we want to take advantage of code similarity to write code that can be used in as many situations as possible. This intuition leads us to writing generic code, code

that is highly parameterized so that it can be easily reused in a wide variety of situations.

Details of generic code are supplied at compile time or run time instead of hard coding them. Any code that has parameters, such as function calls, can be considered generic, but the term is usually reserved for code built around templates (in C++) or parameterized classes (in SystemVerilog). Making programs generic is consistent with the OOP goal of separating concerns. Thus OOP languages provide facilities for building generic code.

A parameterized class is one that (obviously) has parameters. The syntax in SystemVerilog for identifying parameters is a pound sign (#) in the class header followed by a parenthesized list of parameters. As an example, consider the following parameterized class:

```
class param #(type T=int, int R=16);
endclass
```

This class has two parameters, `T`, which is a type parameter and `R`, which is an integer parameter. Instances of a parameterized class with specific values for the parameters create specializations, that is, versions of the code with the parameters applied.

```
param #(real, 29) z;
param #(int unsigned, 12) q;
```

The above declarations create specializations of the parameterized class `param`. The class name and parameters identify specializations. Thus, specializations are in fact, unique types. The compiler will not allow you to assign `q` to `z`, or vice versa, because they are objects of different types.

`type` parameters allow you to write type-independent code, code whose data structures and algorithms can operate on a wide range of data types. For example:

```
class maximizer #(type T=int);
   function T max(T a, T b);
      if( a > b )
           return a;
      else
           return b;
   endfunction
endclass
```

The parameterized class `maximizer` has a function `max()` that returns the maximum of two values. The `max` algorithm is the same no matter the type of

the comparison objects. In this case, the only restriction is that the objects be comparable with the greater than (>) operator.

Classes cannot be meaningfully compared using the greater-than operator, so a different version of maximizer is necessary to deal with classes. To make a version of maximizer that will return the largest of two class objects, we must define a method in each class that will compare objects.

```
class maximizer #(type T=int);
   function T max( T a, T b);
      if( a.comp(b) > 0 )
            return a;
      else
            return b;
   endfunction
endclass
```

This presumes that the type parameter T is really a class, not a built-in type, such as int or real. Further, it presumes that T has a function called comp(), which is used to compare itself with another instance. The OVM library contains a parameterized component called ovm_in_order_comparator#(T), which is used to compare streams of transactions. It has two variants, one for comparing streams of built-in types, and one for comparing streams of classes. The reason we need two in-order comparator classes is exactly the same reason we need two maximizers—SystemVerilog does not support operators that can operate on either classes or built-in types.

2.5.1   Generic Stack

Our stack is not particularly generic. It has a fixed stack size of 20, and the data type of the items kept on the stack is fixed to be int. Below is a more generic form of stack that changes these fixed characteristics to parametrized characteristics.

```
53    class stack #(type T = int);
54
55       local T stk[];
56       local int stkptr;
57       local int size;
58       local int tp;
59
60       function new(int s = 20);
61          size = s;
62          stk = new [size];
63          clear();
64       endfunction
65
```

```
66      function bit pop(output T data);
67
68        if(is_empty())
69          return 0;
70
71        data = stk[stkptr];
72        stkptr = stkptr - 1;
73        return 1;
74
75      endfunction
76
77      function bit push(T data);
78
79        if(is_full())
80          return 0;
81
82        stkptr = stkptr + 1;
83        stk[stkptr] = data;
84        return 1;
85
86      endfunction
87
88      function bit is_full();
89        return stkptr >= (size - 1);
90      endfunction
91
92      function bit is_empty();
93        return stkptr < 0;
94      endfunction
95
96      function void clear();
97        stkptr = -1;
98        tp = stkptr;
99      endfunction
100
101     function void traverse_init();
102        tp = stkptr;
103     endfunction
104
105     function int traverse_next(output T t);
106        if(tp < 0)
107          return 0; // failure
108
109        t = stk[tp];
110        tp = tp - 1;
111        return 1;
112
113     endfunction
114
115     virtual function void print(input T t);
116        $display("print is unimplemented");
117     endfunction
118
```

```
119      function void dump();
120
121        T t;
122
123        $write("stack:");
124        if(is_empty()) begin
125          $display("<empty>");
126          return;
127        end
128
129        traverse_init();
130
131        while(traverse_next(t)) begin
132          print(t);
133        end
134        $display();
135
136      endfunction
137
138   endclass
file: 02_intro_to_OOP/02_generic_stack/stack.sv
```

The generic `stack` class is parameterized with the type of the stack object. The
parameter `T` contains a type. In this case, `T` can be either a class or a built-in
type because we are not using operators directly on objects of type `T`. Any
place in the class where we previously used `int` as the stack type, we now use
`T`. For example, `push()` now takes an argument of type `T`. Class parameters,
such as `T`, are compile-time parameters, meaning the value is established at
compile time. To specialize `stack#(T)`, we instantiate it with a specific value
for the type. For example:

```
stack #(real) real_stack;
```

This statement creates a specialization of `stack` that uses `real` as the type of
object on the stack.

The size of the stack is no longer fixed at 20. We use a dynamic array to store
the stack, whose size is specified as a parameter to the constructor. Unlike `T`,
the argument `size` is a run-time parameter—its value is specified when the
program runs. This lets us create multiple stacks, each with a different size.

```
stack #(real) big_stack;
stack #(real) little_stack;

...

big_stack = new(2048);
little_stack = new(6);
```

`big_stack` and `little_stack` are of the same type. They use the same specialization of `stack#(T)`. However, they are each instantiated with different size parameters.

In making `stack` generic, we made another change. We replaced `dump()` with `traverse_init()` and `traverse_next()`. `dump()` relies on the type of the stack elements, which is not known until compile time. We need to be able to traverse the stack and format each element no matter what the element type is. It could be an `int`, or it could be a complex class with multiple members. We don't know what it will be. To keep `stack#(T)` generic, we must resist all temptation to establish any reliance on the type of the stack elements.

Whereas `dump()` will run through the stack elements and print them in order, `traverse_init()` sets an internal traversal pointer (`tp`) to point to the top of the stack, and `traverse_next()` hands the current element (as pointed to by `tp`) back to the caller and decrements `tp`. The stack maintains some state information about the traversal. The state information is reset when `traverse_init()` is called.

By making `stack#(T)` generic, removing reliance on hardcoded types and sizes, we have made this component highly reusable.

2.6   Classes and Modules

Interestingly, HDLs, such as Verilog and VHDL, though not considered object-oriented languages, are built around concepts quite similar to classes and objects. Module instances in Verilog, for example, are objects, each with its own data space and set of tasks and functions. Just like objects in OO programs, each instance of a module is an independent copy. All instances share the same set of tasks and functions and the same interfaces, but the data contained inside each one is independent from all other instances. Modules are controlled by their interfaces. Verilog modules do not support inheritance (that is, the ability to form IS-A relationships) or type parameterization, and they are static, which makes them unsuitable for true OOP.

The similarity between classes and modules opens up an opportunity for us to use class objects in a hardware context. We can create verification components as instances of classes, giving us the flexibility of classes along with the connection to hardware elements. The designers of SystemVerilog have capitalized on this relationship when extending Verilog with classes, providing the capability for a class to work a lot like modules.

The table below compares features of classes in Verilog, SystemVerilog, and C++.

Feature	Verilog Modules	C++ Classes	SystemVerilog Classes
local data space	yes	yes	yes
function interface	kind of	yes	yes
port interface	yes	no	yes
inheritance	no	yes/multiple	yes/single
type parameterization	no	yes	yes
dynamic	no	yes	yes

The SystemVerilog feature that makes this possible is the *virtual interface*. A virtual interface is a reference to an interface (here we refer to the SystemVerilog interface construct). We can write a class containing references to items inside an interface that doesn't yet exist (that is, it isn't instantiated). When the class is instantiated, the virtual interface is connected to a real interface. This makes it possible for a class object to both drive and respond to pin activity. SystemC modules are implemented as classes and allow for pins to be in the port list, providing the same sort of structure.

HDLs, such as Verilog and VHDL, lack many OOP facilities, and thus are not well suited for building testbenches. The fundamental unit of programming in most HDLs is the module, which is a static object. Modules come into existence at the very beginning of the program and persist unmodified until the program completes. They are syntactically static as well—the syntactic means to modify a module to create a variant are limited. Verilog allows you to parameterize scalar values, but not types. Often you are reduced to cutting and pasting code, then making local modifications. If you have ten different variations you need in a particular design, you must paste ten copies in appropriate locations and then locally modify each one. Should the template module change (the one that you pasted around to create the variants), you'll have to locate each instance and make those same changes in each one. This process is not all that different from what our assembly language programmers had to do fifty years ago.

Sidebar: Simula 67

The relationship between class objects and hardware simulation has been around for quite some time. Simula 67,[1] one of the earliest OOP languages, was developed explicitly for the purpose of building discrete event models. Simula 67 has the notion of class objects and a simulation kernel. It even has a kind of PLI for connecting in external Fortran programs. Simula provides DETACH and RESUME keywords, which allow processes to be spawned and reconnected, sort of a fork/join. It has a special built-in class called SIMULATION, which provides event list features.

Even though the terms object and object-oriented are not used at all in Simula 67, all modern object-oriented programs can trace their lineage to this early programming language. Discrete event simulation languages also can trace their genesis to Simula 67. For many, bringing together the ideas of OOP and hardware simulation seems new; but in fact, the two ideas were born together and only later parted ways. Using OOP with a discrete event simulator brings us full circle.

According to Ole-Johan Dahl and Kristen Nygaard, Department of Informatics, University of Oslo:[2]

> Simula 67 still is being used many places around the world, but its main impact has been through introducing one of the main categories of programming, more generally labelled object-oriented programming. Simula concepts have been important in the discussion of abstract data types and of models for concurrent program execution, starting in the early 1970s. Simula 67 and modifications of Simula were used in the design of VLSI circuitry (Intel, Caltech, Stanford). Alan Kay's group at Xerox PARC used Simula as a platform for their development of Smalltalk (first language versions in the 1970s), extending object-oriented programming importantly by the integration of graphical user interfaces and interactive program execution. Bjarne Stroustrup started his development of C++ (in the 1980s) by bringing the key concepts of Simula into the C programming language. Simula has also inspired much work in the area of program component reuse and the construction of program libraries.

---

1. Lamprecht, Gunther, "Introduction To Simula 67," Vieweg, 1983
2. http://heim.ifi.uio.no/~kristen/FORSKNINGSDOK_MAPPE/F_OO_start.html

2.7   OOP and Verification

Building an object-oriented program and building a testbench are not very different things. A testbench is a network of interacting components. OOP deals with defining and analyzing networks of interacting objects. Objects can be related through IS-A or HAS-A, and they communicate through interfaces. OOP just naturally fits the problem of building testbenches.

Languages such as SystemC/C++ and SystemVerilog, which do provide OOP facilities, are better suited for testbench construction than HDLs, such as Verilog and VHDL. Using dynamic classes, parameterized classes, inheritance, and parameterized constructors, you can build components that are flexible, reusable, and robust. Spending a little extra time to build a generic component can result in a large productivity gain when that component is reused in different ways in many places.

Springer