

## Chapter 8

# Advanced OOP and Testbench Guidelines

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

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

Sometimes it is difficult to divide the functionality into separate parts. Take the example of a bus transaction with error injection. When you write the original class for the transaction, you may not think of all the possible error cases. Ideally, you would like to make a class for a good transaction, and later add different error injectors. The transaction may have data fields and an error-checking CRC field generated from the data. One form of error injection is corruption of the CRC field. If you use composi-