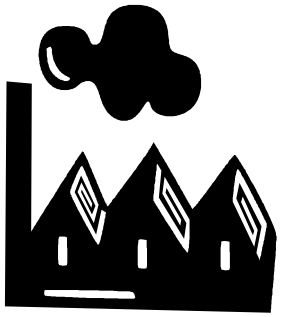
5 USING THE FACTORY PATTERN



This chapter describes the factory pattern to design highly reusable testbenches. The factory pattern allows the top-level test to change the pattern without having to rewrite the code, just as a widget factory can stamp out new parts without having to install new machines. Factory pattern is a well known concept in Object Oriented programming and can be adopted to provide additional flexibility in the choice of classes to be used in the verification. To demonstrate this concept, we use a factory pattern in two examples. In the first example we are using an atomic generator but we select a transaction instance with different constraints than what was defined in the basic transaction class. In the second example we are using a command transactor that uses an error injection object to randomly force errors. However, we are modifying in the testcase the instance that was defined in the environment. Chapter 6 also uses the second example to demonstrate the callback pattern.

5.1 FACTORY DEFINITION

The term "factory" originates from the car manufacturing process where every car being built has the same set of base class of objects being installed, such as "radio", "seat", "engine", etc. However, each car being manufactured has a different type of "radio" (e.g., AM/FM with DVD, or AM only, or AM / satellite / cell phone), or different type of engine (e.g., 6 cylinders, 8 cylinders, etc). A factory pattern is used to accommodate a manufacturing process that calls for the installation of radios, engines, seats, etc, but yet to allow individual selection of the items being installed. The base software calls for the installation of these items; however, the actual choice of the objects to be installed is deferred in the software via a link that relates to the purchase order.

In Object-Oriented programming, a factory pattern is a well known technique for creating an object. It lets a subclass decide which class to allocate, thus deferring the allocation of the class to subclasses.[1] For example, a generator designed to generate basic Ethernet L2 packets can be later reused to create TCP-IP packets with very little modification. Without the factory pattern technique, such a change would require extensive change to already known-to-work code, which is generally discouraged. A typical use of VMM factories is in transactors that generate transaction objects. For examples, factories can be used to do the following:

- Select a class with a specific error injection algorithm.
- Select a transaction class with different constraints.
- Select a class with additional coverage.
- Select a class with different reporting procedures.

The advantage of a factory approach is that the original implementation of the transactor remains unchanged even though it can create very different objects. The behavior of the transactor can be entirely different that what was implemented by default originally. Such flexibility is required in verification to leverage on stable, working code, yet be able to tailor to specific testcase needs. Such a change can be done at the *program* block level without changing the underlying transactor or environment level. For example, you can change the behavior of a transaction generator by modifying the sets of transaction constraints to use. Such flexibility comes with a well designed base transactor. The design and coding guidelines of such a factory-based transactor is illustrated in the next sections.

5.2 FACTORY EXAMPLE – CONSTRAINTS

Consider the case where you defined a transaction class with a set of constraints, an atomic generator (e.g., with the *`vmm_atomic_gen*), and an environment instantiated in a program. You simulated the design and obtained a set of coverage metrics; now you want to rerun the model with a different set of constraints. To achieve this goal you need to extend your transaction class. But the question then becomes, "how big of a change is this?" Do you need to create a new generator and a new set of interconnections in the environment? Is this a big change? **NO!** VMM flow control and the factory pattern used by the atomic generator are designed such that it allows you to make the testcase redefinition in the *program* block without modifying the environment. To achieve this flexibility, certain rules must be observed in the design of the classes.

In the FIFO example, we define a base *FIFO_xactn* class with a set of constraints. We then create an atomic generator using the macro *`vmm_atomic_gen(Fifo_xactn, "FIFO Xaction*

[1] For more information on using factories in Object Oriented programming, refer to *Design Patterns: Elements of Reusable Object-Oriented Software* (Addison-Wesley Professional Computing Series) by Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides ISBN: 0201633612.

Generator"). While this basic environment will be a good fit for random PUSH, POP kind of transactions, to write a focused testcase with PUSH transactions alone, you need a transaction with stringent constraints. We also define two class extensions of *Fifo_xactn* called *Fifo_xactn_no_push* and *Fifo_xactn_no_pop*, each with different sets of constraints, as shown in Figure 5.2-1.

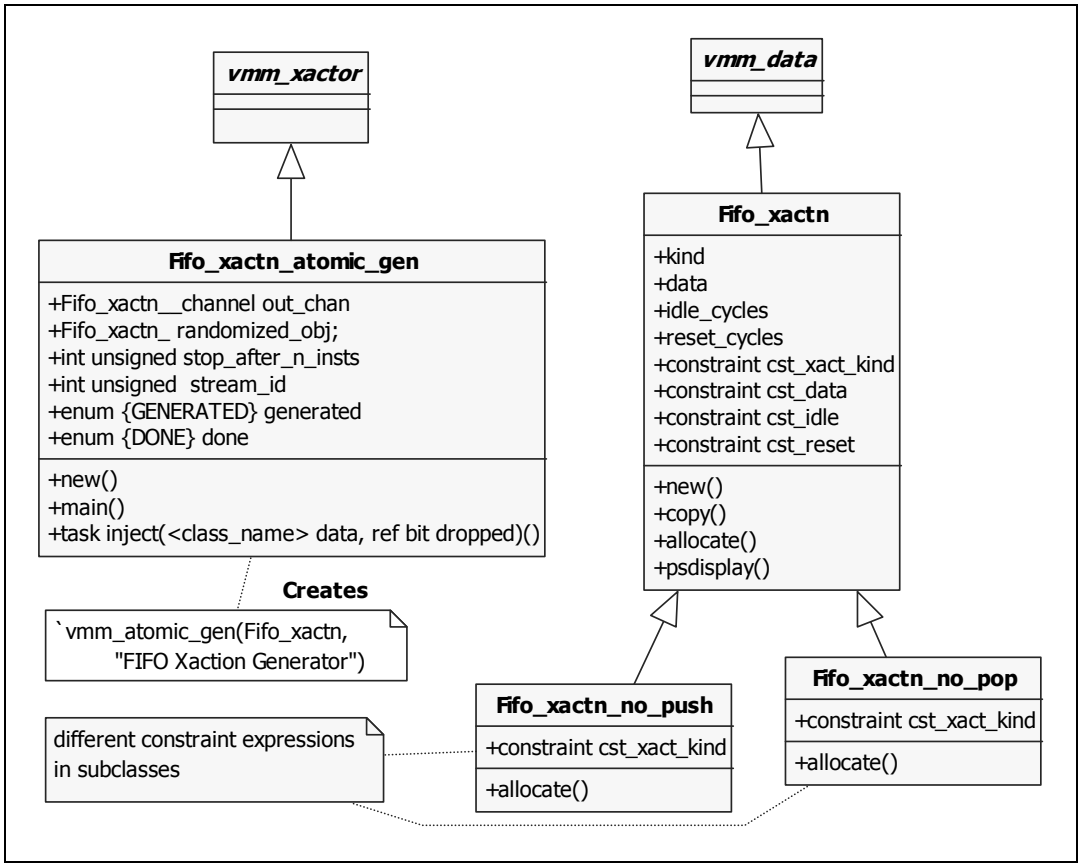


Figure 5.2-1 UML for Transactions and Atomic Generator Classes

In addition to the constraints, we need to define the *copy* method.[2] Generators use *copy()* while monitors use *allocate().* The *vmm_data::allocate()* method is simply a call to the *new()* method and appears redundant. But, it enables the creation of factories and the use of polymorphism in transactors, which is not possible with the direct use of the constructor. *Fifo_xactn_no_pop* class is shown in Figure 5.2-2.

```
class Fifo_xactn_no_pop extends Fifo_xactn;
  constraint cst_xact_kind {
    kind dist {
      PUSH := 25,
      POP := 0,
      PUSH_POP :=0,
      IDLE := 3,
      RESET := 1
    };
  } // cst_xact_kind
```

[2] VMM Rule 4-76 All classes derived from the *vmm_data* class shall provide implementations for the *psdisplay*(), *is_valid(), allocate(), copy()* and *compare*() virtual methods.

```
 extern virtual function  vmm_data copy(vmm_data to=null);
endclass : Fifo_xactn_no_pop

function  vmm_data Fifo_xactn::copy(vmm_data to);
  Fifo_xactn cpy;
  if (to !=null) begin
    if (!$cast(cpy, to)) begin
      `vmm_fatal(log,
          "Attempting to copy a non fifo_xactn instance");
       return;
     end
  end else cpy =new;
  super.copy_data(cpy);
  cpy.kind = this.kind;
  cpy.data = this.data;
  cpy.idle_cycles = this.idle_cycles;
  copy = cpy;
endfunction : copy
```

Note: If the extended class has the same variables as its base class, then there is no need for a *copy* in the extended class since the *copy* from the base class will be used.

Figure 5.2-2 *Fifo_xactn_no_pop* Example for *Fifo_xactn* (*ch5_fct_xactn /fifo_xactn.sv*)

The VMM atomic generator is implemented using a factory pattern so that it is reuse friendly. A factory-based generator can be used for the generation of transactions derived from different transaction descriptors.[3] The property <*class_name*> *randomized_obj* is a transaction or data descriptor instance that is repeatedly randomized to create the random content of the output descriptor stream. The atomic generator uses a factory pattern to generate the output stream instances. The generated stream can be constrained using constraint techniques defined in IEEE P1800, section 13. Figure 5.2-3 demonstrates the environment as it relates to the use of classes. This environment remains unchanged when you want to use a different transaction model, such as the NO POP case in the constraints (see *ch5_fct_xactn/fifo_xactn.sv* for model of the constraint).

```
class Fifo_env extends vmm_env;
  Fifo_xactn_atomic_gen fifo_xactn_gen_0;   // atomic generator declaration
..
function void Fifo_env::build();
  ..
```

***fifo_xactn_gen_0.randomized_obj* is not addressed here. Instead, *fifo_xactn_gen_0.randomized_obj* uses default *Fifo_xactn* type generated during the creation of the generator with the macro**

```
// Instantiation of transaction generator
    this.fifo_xactn_gen_0 = new ("fifo_gen", 0, fifo_channel_0);
…
endfunction : build
```

Figure 5.2-3 The Environment Remains Unchanged
Redefinition Performed at *program* Level (*ch5_fct_xactn/fifo_env.sv*)

[3] VMM book page 127, OOP Primer: Virtual Methods
VMM book page 217, OOP Primer: Factory Pattern

The changes are made in the *program* to redefine the data instance that gets generated, as shown in Figure 5.2-4. Note that the *Fifo_env::build()* is first exercised, thus setting the *fifo_xactn_gen_0.randomized_obj* to a default object of type *FIFO_xactn.* Following the *Fifo_env*::*build(),* you redefine the handle of the *fifo_env_0.fifo_xactn_gen_0.randomized_obj* to the handle of the desired transaction instance:

```
fifo_env_0.fifo_xactn_gen_0.randomized_obj= fifo_xactn_no_pop;
```

Following this redefinition, you then call the *Fifo_env::run* for the remainder of the test.

```
program automatic fifo_test_pgm ();
  timeunit 1ns; timeprecision 100ps;
  //include files + log + fifo_env_0 instantiation
  `include "test.svh"
  initial :test
  begin
  // Build all components of an environment - testbench
     `vmm_note(log,"Start of Test");
      // Do the build first
      fifo_env_0.build();                                          (1)
      // modify the default environment for the fifo_env_0.randomized_obj
      begin : setting_up_the_factory_for_the_generator
        // Declare an instance and instantiate desired transaction with constraints
        Fifo_xactn_no_pop fifo_xactn; // No pop constraint
        // Fifo_xactn_no_push fifo_xactn_no_push; // no push
         fifo_xactn=new();
         `vmm_trace(log,
            "Modifying reference of randomized_obj to NO POP");
        fifo_env_0.fifo_xactn_gen_0.randomized_obj= fifo_xactn;    (2)
      end : setting_up_the_factory_for_the_generator
      // now run the environment.
      // Since the build was exercised already, it will not be repeated in the run
     fifo_env_0.run();
     `vmm_note(log, "End of Test");                                (3)
  end :test
endprogram : fifo_test_pgm
```

Figure 5.2-4 *program* Level (*ch5_fct_xactn/fifo_pgm.sv*)

An interesting question: does the *run()* re-exercise the *build* from the environment, and if it did, wouldn't that redefine our earlier changes? The answer to the first question is NO. Referring to section 4.1.2 "Test Flow Section", the first call to a step within the flow (e.g., *fifo_env_0.build()*) executes all of the preceding steps up to and including the called step (i.e., *gen_cfg()* and *build()* are executed). If another step is called, then the flow continues from where it left off, up to and included the called step. For example, a call to *fifo_env_0.run()* that follows *build()* executes *reset_dut(), cfg_dut(), start(), wait_for_end(), stop(), cleanup(),* and *report()*). It is an error to call a step prior to the last executed step. Thus, in the *program*, if you call *Fifo_env::build(),* then *Fifo_env::gen_cfg(),* an error message will be issued. This stepping forward flow methodology ensures that all tests execute in the proper sequence. It also allows the modification of handles or variables to be executed after the *Fifo_env::build().* The remaining control flow can then continue to the other steps. A simulation of this code with trace ON demonstrates the build process, as shown in Figure 5.2-5. With VCS, trace is turned on with the "+rvm_log_default=trace" command line option.

```
0.00 ns test [Normal:NOTE] | Start of Test
0.00 ns fifo_env [Trace:INTERNAL] | Generating test configuration...
0.00 ns fifo_env [Trace:INTERNAL] | Building verification environment...
0.00 ns fifo_env [Trace:DEBUG] | doing build
0.00 ns fifo_env [Normal:NOTE] | Sim shall run for no_of_xactions 231
0.00 ns fifo_env [Trace:DEBUG] | end of build
0.00 ns test [Trace:DEBUG] | Modifying reference of randomized_obj to NO POP
0.00 ns fifo_env [Trace:INTERNAL] | Reseting DUT...
1950.00 ns fifo_env [Trace:INTERNAL] | Configuring...
1950.00 ns fifo_env [Trace:INTERNAL] | Starting verification environment...
1950.00 ns fifo_env [Trace:INTERNAL] | Saving RNG state information...
1950.00 ns fifo_env [Trace:INTERNAL] | Waiting for end of test...
1950.00 ns FIFO Xaction Generator Atomic Generator [Trace:INTERNAL] | Started
1950.00 ns cmd_xactor [Trace:INTERNAL] | Started
1950.00 ns cmd_xactor [Trace:DEBUG] | Got a new fifo xaction from in_channel  #0.0.0 Fifo
Xaction RESET  Cycles 0
1950.00 ns Fifo Monitor Xactor [Trace:INTERNAL] | Started
2250.00 ns cmd_xactor [Trace:DEBUG] | Got a new fifo xaction from in_channel  #0.0.1 Fifo Xaction PUSH
2350.00 ns Fifo Monitor Xactor [Trace:DEBUG] | Found a PUSH Xactn at time 2350.00 ns data e7
```

Figure 5.2-5 Simulation Run Demonstrating the Build Process

The key points in setting up the factory patterns in the *program* block for modifying transaction objects generated by a transactor:

1. Build the environment *build()*.
   ```
   fifo_env_0.build();
   ```
2. Following the *build()*, declare within a *begin end* block a declaration and an instantiation of the desired transaction object.
   ```
   begin : setting_up_the_factory_for_the_generator
     Fifo_xactn_no_pop fifo_xactn;
     fifo_xactn=new();
   // Must then point handle of desired object to the new allocated handle.
     fifo_env_0.fifo_xactn_gen_0.randomized_obj= fifo_xactn;
   end : setting_up_the_factory_for_the_generator
   ```
3. Continue with the environment *run()* method.
   ```
   fifo_env_0.run();;
   ```

And that's it! If you need to change the choice of transactions with a different set of constraints, just change the ONE line that declares the transaction variable. For example, to select the NO PUSH set of constraints, substitute the transaction class declaration with

```
Fifo_xactn_no_push fifo_xactn; // No push constraints
```

No other changes are required.

5.3 FACTORY EXAMPLE – ERROR INJECTION

Figure 5.3-1 represents the top level view of the testbench. The transactor injects a data error using an algorithm defined in an error injection class. In addition, the conditions needed to inject the error are also in that class. We're interested in selecting one of many algorithms for the scheduling and implementation of the errors.

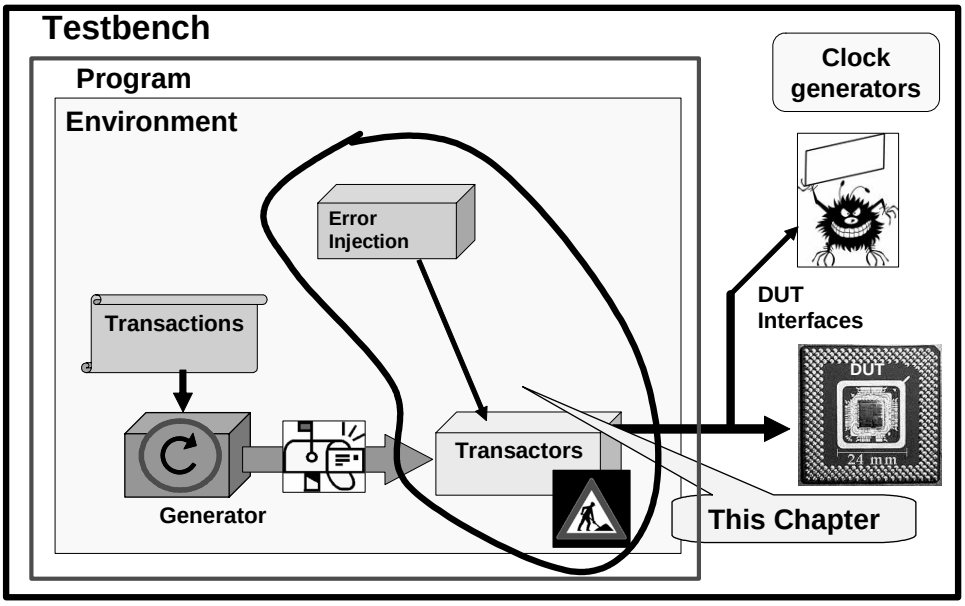


Figure 5.3-1 Test environment with Error Injection

We're demonstrating the use of the factory pattern for an error injection. Note that a factory pattern to inject errors requires that the error injection mechanism be already implemented in the lower level transactors, such as command-layer transactors. If it is not implemented a priori, errors can only be injected using callbacks. However, a callback extension can use a factory pattern to provide controllability over the injected errors

Two decisions must be made for the error injection: When to inject the error, and what error to inject. The factory pattern can help in those decisions because it can be modified from the *program* without having to modify or extend the transactor any further.

The scheduling of the error was not initially specified as an element of the *kind* enumeration scheduled for randomization. This is because, like a real project, it came as a late requirement. Instead, we'll use a variable *inject_err* that is randomized with a constraint only when *kind==PUSH*. If that variable has the value INJECT, then the data to be written is modified with the *flip_data* function defined in the *Data_err_inject* class (or in an extension of that class).

In the file *ch5_fct_inject_err /fifo_pkg.sv* we define an enumeration type

```
typedef enum {INJECT, NO_INJECT} inject_err_t;
```

In class *Inject_err* we define the property *inject_err* along with a constraint to determine the scheduling of the error injection.

```
rand inject_err_t inject_err;
```

The *Inject_err* class represents a template for an error injector class with a default of no errors by setting the constraint to a distribution of NO error injection. It also includes a dummy function to corrupt the data, but is implemented such that, if called, it does NOT corrupt the data. The *Inject_err2* class extends the *Inject_err* class, and provides a constraint for a distribution on the error injection. We also specified the algorithm to corrupt the data. Our simple algorithm flips some data bits. Those classes are shown in Figure 5.3-2.

```
class Inject_err;
  import fifo_pkg::*;
  static vmm_log log = new("Inject_err", "class");
  rand inject_err_t inject_err;
  constraint cst_inject_err{
    inject_err dist {
      INJECT := 0,    // No errors by default
      NO_INJECT :=100
   };
  } // cst_inject_err

  virtual function word_t corrupt_data(word_t data);
     word_t local_data;
     // local_data={{data[WIDTH-1:1], !data[0]}};
     local_data=data;  // no errors
     `vmm_trace(log,
        $psprintf("data=%h, corrupted %h", data, local_data));
     corrupt_data=local_data;
  endfunction : corrupt_data
endclass : Inject_err

// --------------------
class Inject_err2 extends Inject_err;
  constraint cst_inject_err{
    inject_err dist {
      INJECT := 5,
      NO_INJECT :=100
   };
  } // cst_


 virtual function word_t corrupt_data (word_t data);
     word_t local_data;
     local_data={{data[WIDTH-1:2], !data[1], data[0]}};
     `vmm_trace(log,
          $psprintf("data=%h, corrupted %h", data, local_data));
     corrupt_data=local_data;  endfunction : corrupt_data
endclass : Inject_err2
```

Figure 5.3-2 Inject_err Class *(ch5_fct_inject_err/inject_err.sv)*

Since we have two possible implementations of the error injection function *corrupt_data*, we will use a factory pattern to specify which inject error class instance to use. Figure 5.3-3 provides a UML for the class relationships to build a factory for the FIFO command-layer transactor.

The following guidelines and comments were followed to build such a factory pattern:

1. **Define all methods in the factory class as virtual.** This allows for future expansion to access methods defined in this base class.

2. **Define in the command-layer transactor an instance of the base class of the error injector. Instantiate that instance in the *main()* if it was not allocated.**

```
//  (ch5_fct_inject_err /fifo_cmd_xactor.sv)
 class Fifo_cmd_xactor extends vmm_xactor;
    Inject_err factory_inject_err;
    ..
    task Fifo_cmd_xactor::main();
    …
      if (this.factory_inject_err==null)
              this.factory_inject_err=new();
    forever
       // main body of main
    endtask : main
  endclass : Fifo_cmd_xactor
```

Command-layer transactor

3. **Do not change the original environment that worked with the base error injector class. The base error injector class is not used in the environment.**
//(ch5_fct_inject_err/fifo_env.sv)
In the *Fifo_env::build()* :

Instantiate the command-layer transactor

```
this.fifo_cmd_xactor_0 = new("cmd_xactor",
                0,
                `TOP.f_if,
                fifo_channel_0
                );
```

Environment

4. **Define in the *program* do the following:** *(//(ch5_fct_inject_err /fifo_pgm.sv)*

   a. **In the *initial* block, initiate the *build*.**

```
Begin : initial_pgm
    fifo_env_0.build();
```

Program

   b. **In a *begin end* block, declare an object to inject the desired error. Instantiate that new object and assign it to the command transactor instance of the error injector.**

```
begin : factory_4_error_injection
  Inject_err2 inject_err; // ** New error injection
  inject_err=new();
  fifo_env_0.fifo_cmd_xactor_0.factory_inject_err =
                                          inject_err;
end : factory_4_error_injection
```

   c. **Following this, continue with environment control flow with the *run()* method.**

```
   fifo_env_0.run();
 end : initial_pgm
endprogram : fifo_test_pgm
```

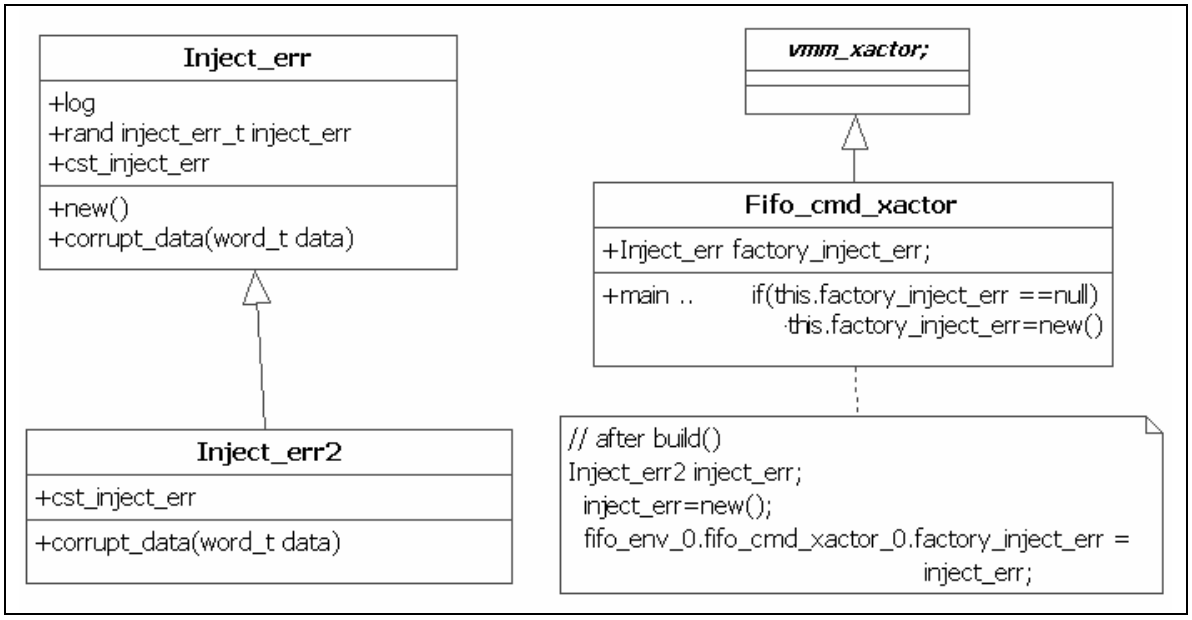


Figure 5.3.-3 UML for the FIFO Command-Layer Transactor

In the command transactor, the *push_task* first randomizes the *factory_inject_err*, and then determines the value to be assigned to the *f_if.driver_cb.data_in* (*data_in* of the virtual FIFO interface clocking block). The *push_task()* is shown in Figure 5.3-4.

```
task Fifo_cmd_xactor::push_task (word_t data);
    f_if.driver_cb.data_in <= data; // default assignment
    if (factory_inject_err.randomize())
      if (factory_inject_err.inject_err==INJECT)
        f_if.driver_cb.data_in <=
                    factory_inject_err.corrupt_data(data);
// ** Common control
   f_if.driver_cb.push <= 1'b1;
   f_if.driver_cb.pop  <= 1'b0;
   @ ( f_if.driver_cb);
   f_if.driver_cb.push <= 1'b0;
  endtask : push_task
```

Figure 5.3-4.push_task with Error Injection *(ch5_fct_inject_err /fifo_cmd_xactor.sv)*

An example of a simulation run displayed the results shown in Figure 5.3-5.

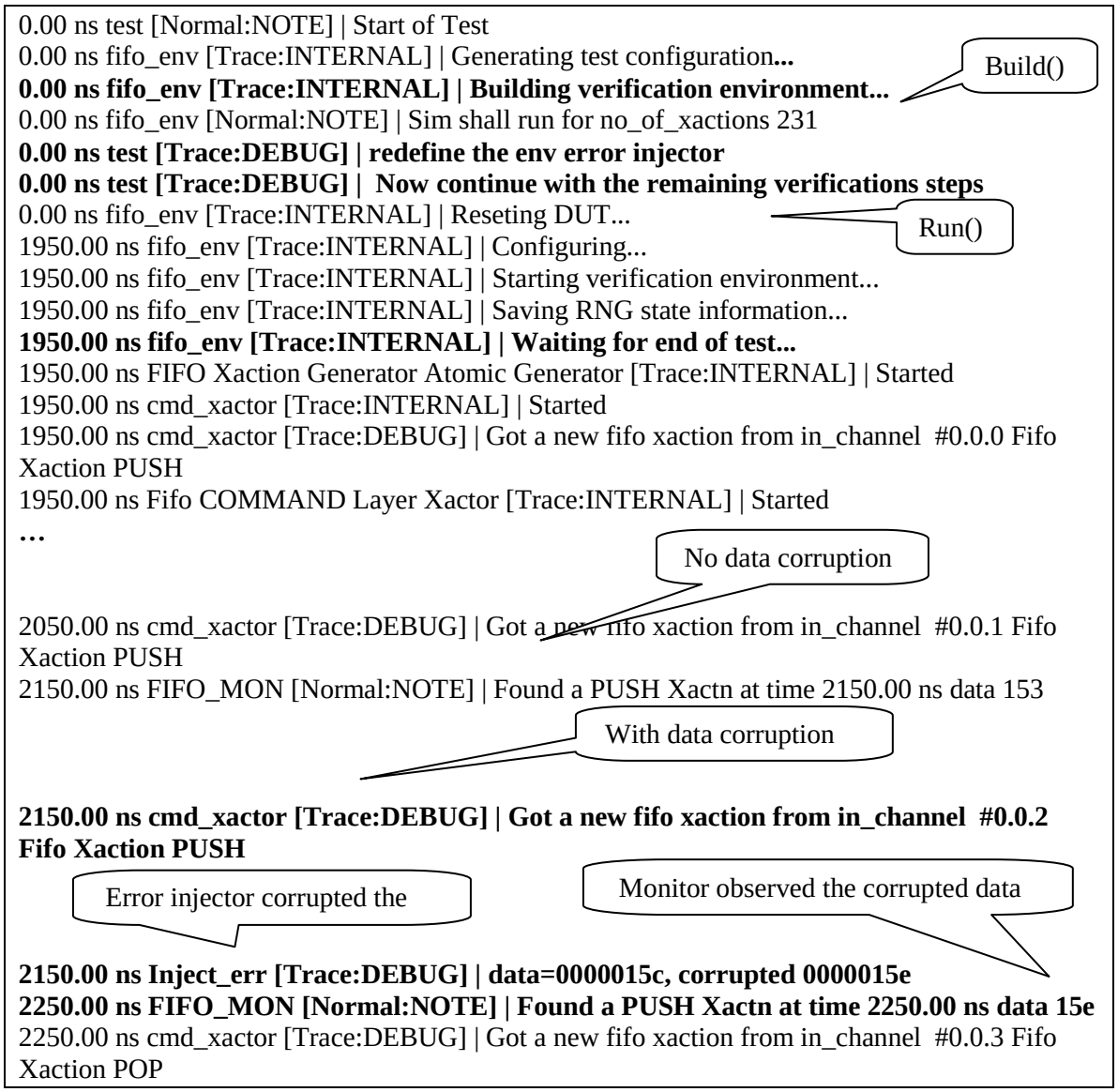


Figure 5.3-5 Simulation Results with Error Injection Using Factory Pattern

5.4 FILE STRUCTURE

Table 5.4 demonstrates the file Structure and the purpose of each file. Figure 5.4 is a graphical representation of the relationships between the files for this chapter.

Table 5.4. File Structure and Functions

/ch5/ch5_fct_xactn and /ch5*/ch5_fct_inject_err* directories

File	Function	Used by
fifo_pkg.sv	Defines types and initialized variables	ALL
fifo_if.sv	Defines the FIFO interface	RTL and by program, testbench, transaction and transactors
fifo_csr_if.sv	Defines the FIFO configuration interface	RTL, property models, and by environment, and possibly transactors
fifo_xactn.sv	Defines the transaction class with the constraints Also used for the channel generation with: *`vmm_channel* (Fifo_xactn)	*`vmm_channel* macro for generation of channel, *`vmm_atomic_gen* macro for generation of atomic generator, monitor transactor for creation of transaction from observed values on bus interface
fifo_rtl.sv	Represents the FIFO RTL DUT	Top level
fifo_props.sv	Defines the properties for assertions	Top level for bind
fifo_log_fmt.sv	Defines formatting information for display	FIFO environment
fifo_pgm.sv	Creates the modeling for simulation and initiates the *run* in the environment	Top level
fifo_env.sv	Creates the build and start for simulation	program
fifo_mon_xactor.sv	Creates a copy of the observed transaction onto a transaction channel.	Scoreboard, top level
top_tb.sv	Represents the top level and instantiates the RTL, the bind, the monitor, etc	none
fifo_gen_xactor.sv	Uses the macro *`vmm_atomic_gen* for generation of atomic generator, defines the constraints for the number of transactions	Environment for creation of the build model
inject_err.sv	Error injection classes	Command transactor
test.svh	Common include files	Program block
fifo_response.sv	Class derived from vmm_data to provide a response to a transactor (e.g., generator) through a channel	command transactor and environment

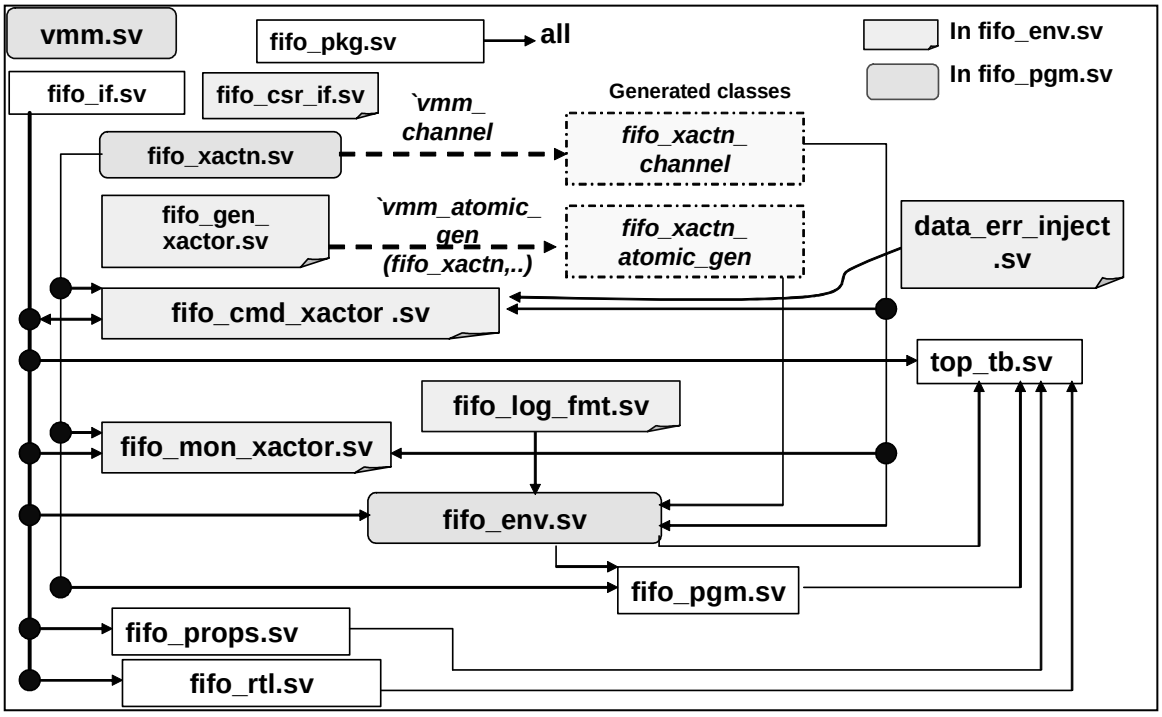


Figure 5.4 Relationships of Files for Factory Pattern

Chapter 5 Questions and LAB

Q1: When should a factory pattern be used in a transactor such as a generator?

Q2. When would you use an atomic generator as created via the macro *`vmm_atomic_gen?*

Q3. Why a custom generator is sometimes needed?

Q4. Why do generators use the *copy* method while monitors use the *allocate* method to create a new object?

Lab05.

Use a factory to select a different transaction class for the randomization of the transactions. See instructions in subdirectory lab/lab05/todo/readme.txt.