Tutorial on ETF: the Eiffel Testing Framework Building a Bank System

Jackie Wang

Fall 2020

Abstract

Customers need not know details of your design of classes and features. Instead, there is an agreed interface for customers to specify the way by which they interact with the software system. The **Eiffel Testing Framework** (**ETF**) addresses exactly this issue, by facilitating software engineers to write and execute high-level, input-output-based acceptance tests.

Customers only need to be familiar with the signature declarations of list of *abstract events*. A first version of ETF that is tailored for the SUD can already be generated, using the ETF generator, using these event declarations (documented documented in a plain text file). This version of ETF, where the actual business model is not yet connected, is already executable on use cases under both batch and interactive modes, where event traces are simply echoed as outputs. User inputs are specified as traces of events (or sequences). Outputs (from executing each event in the input trace) are by default logged onto the terminal, and their formats may be customized. The boundary of the system under development (SUD) is defined by declaring the list of input events (and their parameters) that might occur.

Once the business model is built, there is only a small number of steps to follow for the developers to connect it to the generated ETF. Once connected, developers may re-run all use cases and observe if the expected state effects take place. The design principle of **information hiding** is obeyed here: you as the implementor or designer is free to modify/revise/renew the interior design, and reconnect the new design to the same interface of commands, without affecting the way customers/users interact with the system.

In the generated ETF, many of the Eiffel classes are *deferred* and provides default behaviours that are inherited by the corresponding descendant, effective classes. The ETF user may safely modify these descendant classes, and at some point decide to *renew* the current ETF to re-generate all the deferred classes, as well as the file handling and paring utility classes. That is, any manual changes applied to these descendant classes will persist in all subsequent renewals.

Issues about this tutorial document, or bugs and feature request of the ETF generator should be forwarded to `jackie@eecs.yorku.ca`.

Contents

1 Prerequisites

The command `etf` (an ETF generator) is available on the Prism Linux workstations and also on the SEL Virtual Box image. Confirm this by typing the following on the terminal:

```
% etf -version
Eiffel Testing Framework (ETF) Version: 1.14 (2019-10-12)
% etf -help
The Eiffel Testing Framework (ETF) generator
may be invoked using one of the following 3 options:
     etf -new        defns.txt tar_dir
     etf -renew        defns.txt tar_dir
     etf -help
...
```

This tutorial makes the following assumptions:

– Signatures of events that might occur in your system under development (SUD) is documented in a text file `bank_events.txt`.

– An acceptance test, i.e., a use case or example run of your system, specified in terms of a sequence of event occurrences, is documented in a text file `input.txt`.

2 Generating a new ETF

To generate a new copy of ETF that is tailored to your system (e.g., a bank system), run:

```
etf -new bank_events.txt bank_proj
```

where

1. Option `-new` specifies that a fresh copy of ETF is to be generated.
2. Text file `bank_events.txt` declares
   – Name of the target system (i.e., `bank`)
   – Signatures[1] of events.
     A formal grammar for a system event declaration file such as `bank_events.txt` is included in Section 9.2.1 (p15).

[1]For each event, its signature consists of the event name and a list of parameters (i.e., name-type pairs)

As an example, for a simple bank system, we have file `bank_events.txt` specified as:

```
-- declaration of system name
system bank

-- declaration of event signatures
new(id: STRING)
  -- create a new bank account for "id"
deposit(id: STRING; amount: REAL)
  -- deposit "amount" into the account of "id"
withdraw(id: STRING; amount: REAL)
  -- withdraw "amount" from the account of "id"
transfer(id1: STRING; id2: STRING; amount: REAL)
  -- transfer "amount" from the account of "id1" to that of "id2"
```

where each line of a user comment is written following `--` (i.e., the Eiffel style). Refer to Section 9.3 for the list of supported types for event parameters.

Note. Event names should be unique. Similarly, parameter names of an event should be unique.

3. Target directory `bank_proj` must be existing and empty.
4. Avoid Eiffel keywords for event and parameter names (e.g., `from`).

To see the cluster structure of the generated ETF, refer to Appendix 9.5.

3 Case Studies: Design Patterns in ETF

Each generated ETF project (see Section 9.5) adopts a number of object-oriented design patterns:

– Command Pattern

All system updates are abstracted as *commands*, each of which being modelled as a class. Commands classes (e.g., `ETF_DEPOSIT`, `ETF_WITHDRAW`) constitute a public, external interface, via which customers/users interact with the system. The private, internal design of yours (classes and features) are "connected" to these commands.

Remark. The design principle of **information hiding** is obeyed here: you as the implementor is free to modify the design, and reconnect the new design to the same interface of commands, without affecting the way customers/users interact with the system.

– Singleton Pattern

See the generated classes `ETF_MODEL` (shared data) and `ETF_MODEL_ACCESS` (exclusive access to the shared data).

– Observer Pattern

When system updates are performed via the commands, the changes should be *notified* to the abstract user interface in order to log the latest output. See the `ETF_EVENT` class and all `_COMMAND` classes.

4 Running the Default Generated ETF

From Section 2, we generated a fresh copy of ETF, within the specified target directory `bank_proj`, that is tailored for events declared in the text file `bank_events.txt`. The generated ETF (e.g., one for the simple bank system) is meant to be ready for running (before even implementing your business logic!). As we will see, the default ETF simply echoes the events you input (either from a text file in the batch mode, or from command-line prompts in the interactive mode).

4.1 Building the Executable

To run the default ETF for our simple bank system, we need to first build an executable for it:

```
cd bank_proj
ec19.05 -c_compile -finalize -config bank.ecf
```

Note. Two `.ecf` files are generated for each target system (e.g., `bank.ecf` and `bank-fresh.ecf`):

- When a fresh copy of ETF is generated (via `etf -new ...`), both `.ecf` files are identical.
- When an existing, previously-generated ETF is renewed (via `etf -renew ...`), only `bank-fresh.ecf` will be overwritten, while any user modifications on `bank.ecf` will be kept intact.

  This means that, when changes are needed, apply them *only* to the one (i.e., `bank.ecf`) that will not be overwritten in the subsequent renewals.

4.2 Retrieving the Executable

You may find it convenient to copy the executable built to where the input script `input.txt` is located, e.g., assuming that the current directory is `bank_proj`:

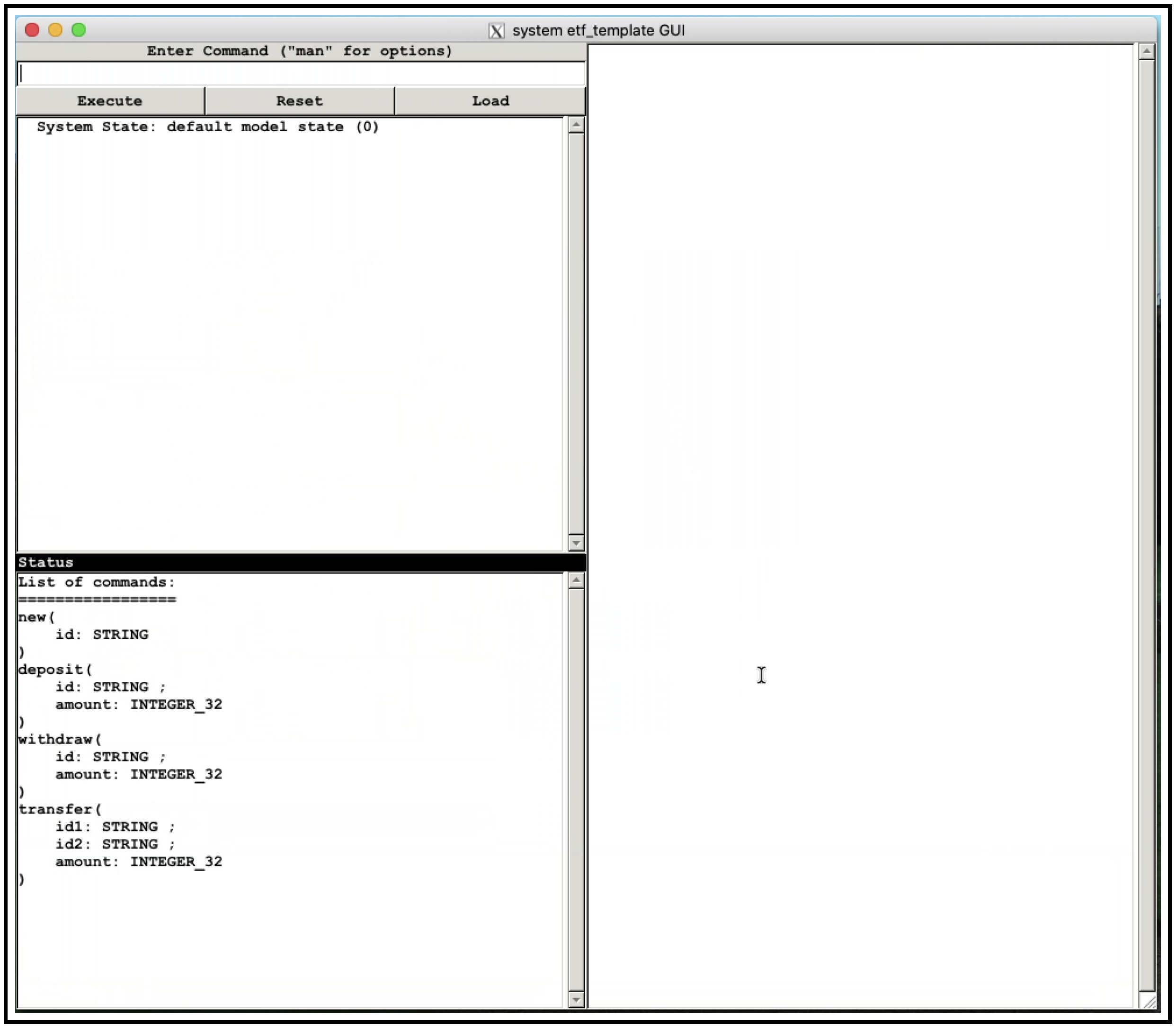
```
cp ./EIFGENs/bank/F_code/bank ./bank.exe
```

4.3 Launching the Default GUI Mode

You may now launch the system:

```
./bank.exe &
```

The default execution mode of a generated ETF project is the GUI mode, as shown in Figure 1.



```
System State: default model state (0)
```

```
List of commands:
=================
new(
    id: STRING
)
deposit(
    id: STRING ;
    amount: INTEGER_32
)
withdraw(
    id: STRING ;
    amount: INTEGER_32
)
transfer(
    id1: STRING ;
    id2: STRING ;
    amount: INTEGER_32
)
```

Figure 1: Graphical User Interface for the Generated Bank ETF Project

The `Status` panel displays commands that correspond to events declared in the source file `bank_events.txt`. From the GUI, you may enter a command (e.g., `deposit("Steve", 200)`), `execute` it, `reset` the system back to its initial state, or `load` a number of commands at once from a text file.

4.4 Changing to the Command-Line Mode

You may often find it inconvenient for your project development in the GUI mode. It is easy switch it:

- Go to the file bank_proj/root/root.e.
- Look for the feature switch and change its value to:

```
switch: INTEGER
    -- Running mode of ETF application
  do
    -- Result := etf_gui_show_history -- GUI mode
       Result := etf_cl_show_history
    -- Result := unit_test -- Unit Testing mode
end
```

- Recompile the project and copy the new version of the executable bank from the EIFGENs directory (see Section 4.1 and Section 4.2).
- Now launching the system again and you will see the difference:

```
% ./bank.exe -help
The executable ./bank.exe may be invoked as:
        ./bank.exe -help
        ./bank.exe -version
        ./bank.exe -i
        ./bank.exe -b     input.txt [output.txt]
        ./bank.exe -w m n input.txt [output.txt]
        ./bank.exe -l     input.txt [output.txt]
```

4.5 Writing Use Cases

We write a use case of running the bank system as a sequence of event occurrences (i.e., an event trace) with argument values conforming to their declared types (in the text file, e.g., bank_events.txt that was used to generate the current ETF).

As an example, here is a valid use case for the bank system:

```
new("Bill")
new("Steve")
deposit("Bill", 55.0)
deposit("Steve", 33.5)
withdraw("Bill", 6.0)
```

Typically we store each use case like the above one into a text file such as input.txt. A formal grammar for a system trace file such as input.txt is included in Section 9.2.2 (p16). Each line must conform to the following syntax:

$$evt\_name \texttt{ (} val_1 \texttt{ , } val_2 \texttt{ , } \ldots \texttt{ , } val_n \texttt{)}$$

That is, each line starts with a string identifier, followed by a left parenthesis, followed by a non-empty comma-separated list of argument values, and ended with a right parenthesis. For writing argument values, refer to Section 9.4.

You may run the above use case in either the batch mode (Section 4.6) or in the interactive mode (Section 4.7). The former has the advantage that the use case is stored in a file and line comments may be used for documentations.

4.6 Running in the Batch Mode

If you have documented a use case of the bank system in a text file `input.txt`, then you may run it in the batch mode:

```
./bank.exe -b input.txt
```

where

1. Option `-b` specifies that the generated bank system is to be run in the batch mode.
2. Text file `input.txt` documents a use case of the bank system, e.g., the one in Section 4.5, with line comments:

```
-- a use case of the bank system

new("Bill")
  -- create a new account named "Bill"
new("Steve")
  -- create a new account named "Steve"
deposit("Bill", 55.0)
  -- deposit $55.0 into Bill's account
deposit("Steve", 33.5)
  -- deposit $33.5 into Steve's account
withdraw("Bill", 6.0)
  -- withdraw $6.0 from Bill's account
```

Consequently, we get the following output from the command line:

```
  System State: default model state (0)
->new("Bill")
  System State: default model state (1)
->new("Steve")
  System State: default model state (2)
->deposit("Bill",55)
  System State: default model state (3)
->deposit("Steve",33.5)
  System State: default model state (4)
->withdraw("Bill",6)
  System State: default model state (5)
```

The line `System State: default model state (0)` is the initial system state. Each symbol `->` is followed by an echo of the input event, and then by the resulting state. For example, the starting state of `new("Steve")` is `System State: default model state (1)`, whereas the resulting state of it is `System State: default model state (2)`.

As no actual business model has been connected to the newly-generated ETF, simply the constant string `System state: default model state` is output for each event. That is, all events, e.g., `new("Bill")` and `deposit("Bill", 55)`, do not have any effect on the bank state.

In Section 6, we will see how to connect your business model implementation (Section 5) to the ETF, so that the expected state changes can be observed.

4.7 Running in the Interactive Mode

You may also run a use case of the bank system "on the fly" by initiating the interactive mode:

```
./bank.exe -i
```

Then you will be prompted to type inputs, which can be either of the following:

- `man` for displaying a manual list of event declarations, as defined in the text file, e.g., `bank_events.txt`, that was used to generate the current ETF;
- an event (e.g., `new("Bill")`, `deposit("Bill", 55.0)`, etc.) that occurs in the current state; and
- `quit` for terminating the interactive mode.

Here is an example run in the interactive mode:

```
indigo 183 % ./bank.exe -i
Enter an event, 'man' for the list of declared events, or 'quit' to terminate...
man
new(id: STRING)
deposit(id: STRING ; amount: REAL)
withdraw(id: STRING ; amount: REAL)
transfer(id1: STRING ; id2: STRING ; amount: REAL)
Enter an event, 'man' for the list of declared events, or 'quit' to terminate...
new("Steve")
  init
->new("Steve")
  System state: default model state
Enter an event, 'man' for the list of declared events, or 'quit' to terminate...
deposit(23, 33.5)
Type Error: specification of command executions is not type-correct
deposit(23, 33.5) does not conform to declaration deposit(id: STRING ; amount: REAL)
Enter an event, 'man' for the list of declared events, or 'quit' to terminate...
deposit("Steve", 33.5)
->deposit("Steve",33.5)
  System state: default model state
Enter an event, 'man' for the list of declared events, or 'quit' to terminate...
quit
```

Notes.

1. The generated ETF is able to report errors when:
   - The line does not have the right syntax. For writing argument values, refer to Section 9.4.
   - The event name does not exist

   - An argument value does not conform its declared type (e.g., value `33.5` for a `STRING` parameter)

2. Similar to the case of running in the batch mode (Section 4.6), since no business model implementation has been connected to the ETF (Section 6), any valid event is only echoed and has no effect on the bank state.

To understand how components of the ETF work internally in the above interactive mode, refer to Section 9.6.

5 Developing Your Own Business Model

In Section 4, we demonstrate how to run use cases on the generated ETF, tailored for the bank system whose events are declared in the text file `bank_events.txt`, both in the batch mode and in the interactive mode. However, all events (and their list of arguments) in an event trace were simply echoed, as there is a lack of an implementation for the bank system.

Appendix 9.7 provides a complete reference implementation for the bank system. All three implementation classes are supposed to be placed under the `model` cluster, where the two generated classes `ETF_MODEL` and `ETF_MODEL_ACCESS` are located. You are expected to study the three Eiffel classes in Appendix 9.7 on your own. Here are some hints:

- A `BANK` has a collection of `ACCOUNTs` (via a client-supplier relationship). Accounts in a bank are indexed by named of owners, using a `HASH_TABLE`.

- The ***singleton design pattern*** is implemented in the `BANK_ACCESS` class to ensure, using the Eiffel `once` routine, that only one instance of the `Bank` class is created.

- Observe the similarity between classes `ETF_MODEL` and `BANK`, and that between classes `ETF_MODEL_ACCESS` and `BANK_ACCESS`. The two classes `ETF_MODEL` and `ETF_MODEL_ACCESS` serve as a template for you to implement the singleton pattern for your business model (e.g., the bank).

Exercise. Complete the implementation for command `transfer` that is missing from the `Bank` class in Appendix 9.7.

Upon completing the implementation of our business model, in the next section we illustrate how to connect it to the generated ETF, so that we will be able to observe the expected state effects of events `new`, `deposit`, `withdraw`, and `transfer` by re-building and re-running the executable `./bank.exe` in both the batch and interactive modes.

6 Connecting Model to the Generated ETF

In principle, you only need to modify classes under the cluster whose name matches that of the SUD (e.g., `bank` as declared in the text file `bank_events.txt`). In the case of our bank example, you need to modify the following files:

1. class `ETF_COMMAND` in `bank/abstract_ui/user_commands/`

   You should declare an attribute of the type of your business model (e.g., `BANK`), so that it is accessible to all the descendant classes (e.g., `NEW`, `DEPOSIT`, etc), and initialize its access in the constructor routine `make`. For example:

```
deferred class
        ETF_COMMAND
inherit
        COMMAND_INTERFACE
        redefine
                make
        end

feature {NONE}
        make(a_name: STRING; a_args: TUPLE; a_container: ABSTRACT_UI_INTERFACE)
                local
                        l_bank_access: BANK_ACCESS
                do
                        Precursor(a_name, a_args, a_container)
                        bank := l_bank_access.bank
                end

feature -- attributes
        bank: BANK
end
```

2. All user-command classes in `bank/abstract_ui/user_command/`

   Now that class `COMMAND` already declares and initializes an access to the business model, each user-command class must use this access to modify the model state, using the corresponding command (i.e., in the `BANK` class). Moreover, after an update is performed, the abstract user interface *container* should be notified in order to output the new system state, using the attached output handler. For example:

```
class
        ETF_DEPOSIT

inherit
        ETF_DEPOSIT_INTERFACE
                redefine deposit end

create
        make

feature
          deposit(a_name:STRING; a_value:INTEGER)
        do
```

```
                        bank.deposit (a_name, a_value)
                        container.on_change.notify ([Current])
        end

    end
```

Apply similar changes to classes ETF_NEW, ETF_WITHDRAW, and ETF_TRANSFER.

Refer to Appendix 9.8 for the Eiffel code listing for NEW and WITHDRAW. The implementation for TRANSFER is left to you as an exercise.

Remark: Each user-command class (e.g., ETF_NEW, ETF_WITHDRAW) is an abstraction of how the system may be modified from the external clients' point of view. But internally, you as the designer has the complete freedom of implementing these abstract commands, via your own classes and features.

3. class `ETF_CMD_LINE_OUTPUT_HANDLER` in `bank/output/`

   See how instances of ETF_MODEL and ETF_MODEL_ACCESS are used in class OUTPUT_HANDLER by default. Replace them with, e.g., BANK and BANK_ACCESS, respectively.

7 Renewing the Generated ETF (Optional)

As you develop the generated ETF, you may decide to generate all _INTERFACE classes with the default behaviours.

```
etf -renew bank_events.txt bank_proj
```

As explained in Appendix 9.5, each class in the *bank* cluster has a parent (deferred) class, in the *generated_code* cluster, that defines the default behaviour. After renewing the current ETF, only those deferred, parent classes will be overwritten. This means that all your changes to the effective, descendant classes will persist. Also, any new classes (outside the ETF) that you created will also persist.

More precisely, for each file in your ETF directory (e.g., bank_proj), exactly one of the following four cases must be satisfied (to see what classes are included in the generated ETF, refer to Appendix 9.5):

– **Case 1: the file is part of the ETF and not meant to be modified**.

  Inspecting the output on the terminal, here is the list of files (mostly located in the *generated_code* and *utilities* clusters) that are reported as being *overwritten* (i.e., any manual changes you made have been wiped out).

– **Case 2: the file is part of the ETF and expected to be modified**.

  All other files (mostly in the *bank* cluster) in the ETF will be left *untouched* (i.e., any manual changes you made will persist).

– **Case 3: the file is not part of the ETF**.

  In this case, the file is left *untouched*. However, caution must be taken to make sure this file does not make the re-generated ETF uncompilable.

– **Case 4: a file as part of the ETF is missing**.

  In this case, a new copy of the file is created.

8 Final Remarks

- If you intend to customize the way output is formatted, modify the ETF_CMD_LINE_OUTPUT_HANDLER class in the *bank/output* cluster.
- Choose action-oriented names for events (e.g., deposit, withdraw, etc.).
- If you intend to modify the parsers, locate and edit the token and grammar specification files:

```
bank_proj/utilities/parse/evt_scanner_def.l        -- scanner specification
bank_proj/utilities/parse/evt_decl_parser_def.y    -- declaration parser specification
bank_proj/utilities/parse/evt_trace_parser_def.y   -- trace parser specification
```

To re-generated the scanner and parser Eiffel classes, locate and run the Windows batch script:

```
bank_proj/utilities/parse/make.bat
```

9 Appendix

9.1 Grammars

We adopt the following notations for presenting the context-free grammars:

– Underscore-separated, all-lower-case compound words (e.g., `type_decl_list`) denote non-terminals.

– Double-quoted words (e.g., `"system"`, `":"`, `","`) or all-capital words (e.g., `IDENT`) denote terminals.

– Each non-terminal rule starts with the name of rule (e.g., `type_decl_list`), followed by a colon (`:`), followed by a number of patterns.

– Each pattern (e.g., `"system" IDENT type_decl_list evt_decl_list`) has a mix of terminals and non-terminals, describing legal strings of the language under specification. Patterns are separated by a vertical bar (`|`), meaning that they are alternatives for substituting the non-terminal they belong to.

– Given a pattern *p*, we adopt the following shorthands:

  `(` *p* `)?` denotes zero or one occurrence/repetition of *p*.

  `(` *p* `)+` denotes one or more occurrences/repetitions of *p*.

  `(` *p* `)*` denotes zero or more occurrences/repetitions of *p*.

– Comments are preceded by `--`.

9.2 Tokens

Token Name	Meaning
`IDENT`	identifier
`STR_LIT`	string literals (within double-quotes)
`CHAR_LIT`	character literals (within single-quotes)
`NUMBER`	unsigned integer literal
`REAL`	unsigned floating-point literal

9.2.1 Grammar for Declaring Abstract System Events

```
declarations ::
          -- declarations of system name and events
          "system" IDENT type_decl_list evt_decl_list
type_decl_list ::
          -- zero or more type declarations
          ( "type" IDENT "=" evt_param_type )*
evt_decl_list ::
          -- zero or more event declarations
          ( IDENT ( "(" evt_param_list ")" )? )*
evt_param_list ::
          ( ":" evt_param_decl )+
evt_param_decl ::
          IDENT ":" evt_param_type
evt_param_type ::
          primitive_param_type
        | composite_param_type
        | IDENT
primitive_param_type ::
          primitive_simple_param_type
        | "TUPLE"
        | "TUPLE" "[" prim_simple_param_list "]"
primitive_simple_param_type ::
          "INT"
        | "INTEGER"
        | ( "-" )? NUMBER ".." ( "-" )? NUMBER
        | "REAL"
        | "VALUE"
        | "BOOL"
        | "BOOLEAN"
        | "CHAR"
        | "CHARACTER"
        | "STRING"
        | "{" enum_item_list "}"
        | IDENT
composite_param_type ::
          "ARRAY" "[" primitive_param_type "]"
prim_simple_param_list ::
          prim_simple_param_decl ( ";" prim_simple_param_decl )*
prim_simple_param_decl ::
          IDENT ":" primitive_simple_param_type
enum_item_list ::
          IDENT
        | enum_item_list "," IDENT
```

9.2.2 Grammar for Defining Use Cases (Event Traces)

```
use_case ::
          -- each use case is specified as a list of event occurrences
          evt_trace
evt_trace ::
          ( IDENT ( "(" evt_arg_list ")" )? )*
evt_arg_list ::
          evt_arg ( "," evt_arg )*
evt_arg ::
          primitive_arg
        | composite_arg
primitive_arg ::
          primitive_simple_arg
        | "[" ( primitive_simple_arg_list )? "]"
primitive_simple_arg ::
          "TRUE"
        | "FALSE"
        | CHAR_LIT
        | STR_LIT
        | NUMBER
        | "-" NUMBER
        | REAL
        | "-" REAL
        | IDENT
primitive_simple_arg_list ::
          -- one or more arguments
          primitive_simple_arg ( "," primitive_simple_arg )*
composite_arg ::
          "<<" ( primitive_arg_list )? ">>"
primitive_arg_list ::
          -- one or more arguments
          primitive_arg ( "," primitive_arg )*
```

9.3 Supported Types

To generate a new ETF (or to renew an existing one), the current generator supports the following types for declaring event signatures in file `bank_events.txt`:

$\boxed{\textit{Primitive Types}}$

- `BOOLEAN`
- `CHARACTER`
- `INTEGER`
- `REAL`
- `STRING`

$\boxed{\textit{Tuple Types}}$

- `TUPLE[`id_1 `:` T_1 `;` id_2 `:` T_2 `;` $\ldots$ `;` id_n `:` T_n`]`

  where

  - each id_i ($i \in n$, $n \geq 1$) is an identifier (following the Eiffel convention)
  - each T_i is a primitive type

  e.g., `TUPLE[acc_id: STRING; amount: REAL]`

$\boxed{\textit{Array Types}}$

- `ARRAY[`T`]`

  where T is either a primitive type or a tuple type

  e.g., `ARRAY[BOOLEAN]`, `ARRAY[TUPLE[acc_id: STRING; amount: REAL]]`

  Note. Each item of a tuple must be specified with a name (i.e., type `TUPLE[STRING; REAL]` will result in a syntax error).

9.4 Supported Value Expressions

To run the generated ETF (in either the interactive mode or the batch mode), you may write expressions (as event arguments) for the list of types specified in Section 9.3:

$\boxed{\textit{Primitive Values}}$

- `true`, `false`, `TRUE`, `FALSE`
- `' '`, `'2'`, `'a'`, `'@'`, `'.'`
- `0`, `-1`, `1`
- `0.0`, `-1.2`, `3.4`, `3.`

  Note. The fractional part of a real number may be unspecified. Integer values are *not* coerced automatically into real values.
- `""`, `" "`, `"ETF"`, `"ETF@york#"`

$\boxed{\textit{Tuple Values}}$

- `["bill", 33.34]`, `["Steve", -54.789]`

$\boxed{\textit{Array Values}}$

- `<<["bill", 33.34], ["Steve", -54.789]>>`

9.5 Overview of the Generated ETF Structure

We briefly describe the generated ETF by following its cluster structure (Figure 2):

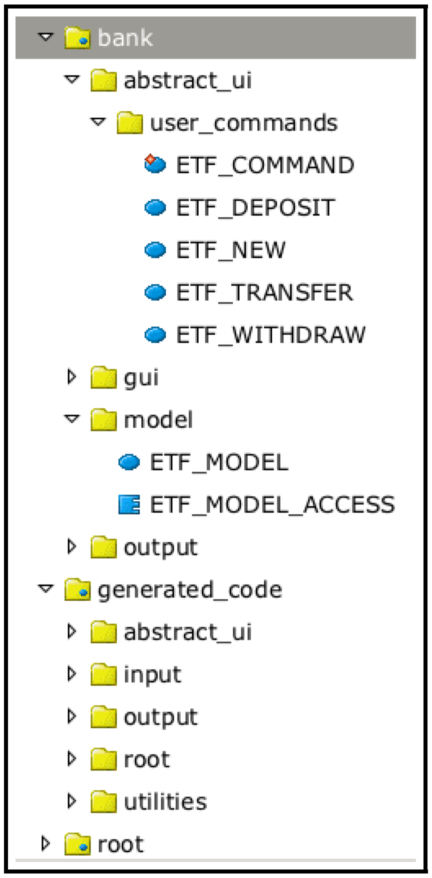

```
bank
  abstract_ui
    user_commands
      ETF_COMMAND
      ETF_DEPOSIT
      ETF_NEW
      ETF_TRANSFER
      ETF_WITHDRAW
  gui
  model
    ETF_MODEL
    ETF_MODEL_ACCESS
  output
generated_code
  abstract_ui
  input
  output
  root
  utilities
root
```

Figure 2: Clusters in ETF

1. Cluster *bank*

   This cluster contains classes that should be modified in order to connect the user business logic to the ETF.

   Most of the classes in this cluster (e.g., `ETF_COMMAND`, `ETF_DEPOSIT`, etc.), when generated, have their default behaviour inherited from the corresponding interface classes (i.e., `ETF_COMMAND_INTERFACE`, `ETF_DEPOSIT_INTERFACE`, etc.) in the *generate_code* cluster.

   1.1 Cluster *abstract_ui*

   Classes in this cluster implement an abstract user interface for the `bank` system.

   1.1.1 Cluster *user_commands*

   Each user-command class corresponds to an event declared in the text file `bank_events.txt` that you used to generate the current ETF (this is an instance of the command design pattern). As detailed in Section 6, you as an ETF user are expected to make changes to classes in the *user_commands* cluster in order to:

   – declare access to the model (in the `ETF_COMMAND` class)

   – update the system state and notify the abstract user interface for logging the output (in the user-command classes, e.g., `ETF_DEPOST`, `ETF_TRANSFER`, etc.)

1.2 Cluster *model*

There are two sample classes `ETF_MODEL` and `ETF_MODEL_ACCESS` that implement a singleton pattern for accessing the model state.

1.3 Cluster *output*

There is a class `ETF_OUTPUT_HANDLER` (a descendant class of `ETF_OUTPUT_HANDLER_INTERFACE` in the *generate_code* cluster) which allows the ETF users to customize how the state-changing commands and (initial and current) states of the model are logged.

2. Cluster *generate_code*

In principle, you do not need to modify classes in this cluster. However, as most classes in the *bank* cluster inherit from the classes here, you are expected to study the behaviour of each of them.

2.1 Cluster *abstract_ui*

Class `ETF_SOFTWARE_OPERATION` inherits from `SOFTARE_OPERATION_INTERFACE`. The main feature in `ETF_SOFTWARE_OPERATION_INTERFACE` is **`execute`** which runs, through the abstract user interface, the list of input commands (passed as a string in both the interactive and batch modes).

2.1.1 Cluster *user_commands*

2.2 Cluster *input*

This cluster contains two classes `INPUT_HANDLER` and `INPUT_HANDLER_INTERFACE`; the former is a descendant class of the latter. In `INPUT_HANDLER_INTERFACE`, there are utility routines that parse and validate the input commands (given as a string), and that convert validated input string into instances of the `COMMAND` class.

In principle, you do not need to override the inherited behaviour from `INPUT_HANDLER_INTERFACE`. As a result, we do not place class `INPUT_HANDLER` in the *bank* cluster.

2.3 Cluster *output*

3. Cluster *root*

There is only the `ROOT` class in this cluster. The **`switch`** feature allows you to run your ETF project in different modes. When the mode is set to **`unit_test`**, you may add tests inside the feature **`add_tests`**.

4. Cluster *utilities*

This cluster contains all utility classes.

4.1 Cluster *event*

There are classes for reading the contents of a text file (i.e., class `ETF_FILE_UTILITY`), and for implementing an observer pattern (i.e., class `ETF_EVENT`).

4.2 Cluster *parse*

There are classes for parsing traces (or sequences) of events: classes `ETF_EVT_TOKENS` and `ETF_EVT_SCANNER` implementing a lexical scanner for the input string, and class `ETF_EVT_TRACE_PARSER` defining a context-free parser for the tokens (created by the scanner).

4.2.1 Cluster *event_arg*

There are classes defining the data type of the input event trace (or sequence).

9.6 A Use Case of Running ETF in the Interactive Mode

To understand how components of the ETF work together, we use the UML sequence diagram (Figure 3, p.23) to illustrate an execution of ETF in its interactive mode (Section 4.7). You are also expected to reproduce this scenario in the debugging mode of Eiffel Studio.

Executing ETF in its interactive mode involves (via option `-i`) the following steps[2]:

1. *Context:* `ROOT.handle_interactive_mode`

   A new interaction of the main loop starts by prompting the user to enter an input (i.e., an event with proper argument values, such as `deposit("Bill", 33.3)`, `man` for the list of previously-declared events, or `quit` to exit from the loop).

2. *Context: Terminal/Console*

   The user enters a previously-declared event (e.g., `deposit`) with valid argument values (e.g., `"Bill"` and `33.3`) and hits return from the terminal. In later steps, we refer to this input string of events as `input_str`.

3. *Context:* `ROOT.handle_interactive_mode`

   An auxiliary procedure `ROOT.exec` is called, where an instance `sys` of class `ETF_SOFTWARE_OPERATION` is created. The procedure `execute` is called upon `sys`, passed with `input_str`.

4. *Context:* `ETF_SOFTWARE_OPERATION.execute`

   This is the main control for the state-changing effects of input event(s) to take place.

   There are five sub-steps of this main control, the first four of which are:

   4.1 Two instances `ui` (of type `ETF_ABSTRACT_UI`) and `output` (of type `ETF_CMD_LINE_OUTPUT_HANDLER`) are created, and the output logging facility (i.e., `output.log_command`) is attached to `ui`.

   4.2 An instance `input` of class `ETF_INPUT_HANDLER` is created. Then `input` is passed with `ui` and `input_str`. The input string will later be converted into a sequence of instances of class `ETF_COMMAND` and added into `ui`.

   4.3 The error reporting facility (i.e., `output.log_error`) is attached to `input`.

   4.4 The procedure `parse_and_validate_input_string` is called upon `input` to see if there are any input errors to report (in which case the flag `input.error` is set to *true*).

5. *Context:* `ETF_INPUT_HANDLER.parse_and_validate_input_string`

   An instance `trace_parser` of class `ETF_EVT_TRACE_PARSER` is created to parse `input_str`. Since the value of `input_str` (i.e., `"deposit("Bill", 33.3)"`) has no syntax error, the attribute `evt_trace` (of type `ARRAY[TUPLE[name: STRING; args: ARRAY[EVT_ARG]]]`[3]) is properly set to represent the input string. More precisely, `trace_parser.evt_trace` is set to

   ```
   <<["deposit", <<"Bill", 33.3>>]>>
   ```

   Then procedure `find_invalid_evt_trace` is called upon `input` to find type errors (e.g., unknown event names, a string argument value for an integer parameter, etc.). Since `deposit` is the name of a declared event, and `"Bill"` and `33.3` are valid with respect to its declaration (i.e., `deposit(id: STRING; amount: REAL)`), no type errors are reported.

   Provided that `evt_parser.evt_trace` contains valid events and argument values, the query `evt_to_cmd` is called upon `input` to produce instance(s) of class `ETF_COMMAND` and inserted into `ui`.

   Then the thread of control returns back to where we left off in `ETF_SOFTWARE_OPERATION.execute`.

[2]these steps form the body of an infinite loop, unless the user enters `quit` to exit.

[3]Class `ETF_EVT_ARG` is located in the *utilities* cluster.

6. *Context:* `ETF_SOFTWARE_OPERATION.execute`

   This is the last, fifth sub-step of the main control.

   The procedure `run_input_commands` is called upon `ui`, which has been properly inserted with the list of commands that are converted from `input_str` in the previous step.

7. *Context:* `ETF_ABSTRACT_UI.run_input_commands`

   For each `COMMAND` instance that is inserted into `ui`, we retrieve its attribute `routine` (declared of type `ROUTINE[ANY, TUPLE]`) and apply its state effect. Due to dynamic binding, the right `ETF_COMMAND` instance will be called upon for its state-changing command (i.e., `ETF_DEPOSIT.deposit`). After the state effect is applied to the model (i.e., the bank system), the corresponding `ETF_COMMAND` instance notifies `ui` about this change so that the command and the updated state are logged.

   Then the current iteration completes, and the thread of control returns back to the terminal.

8. *Context: Terminal/Console*

   The user enters `quit` to exit from the interactive mode.

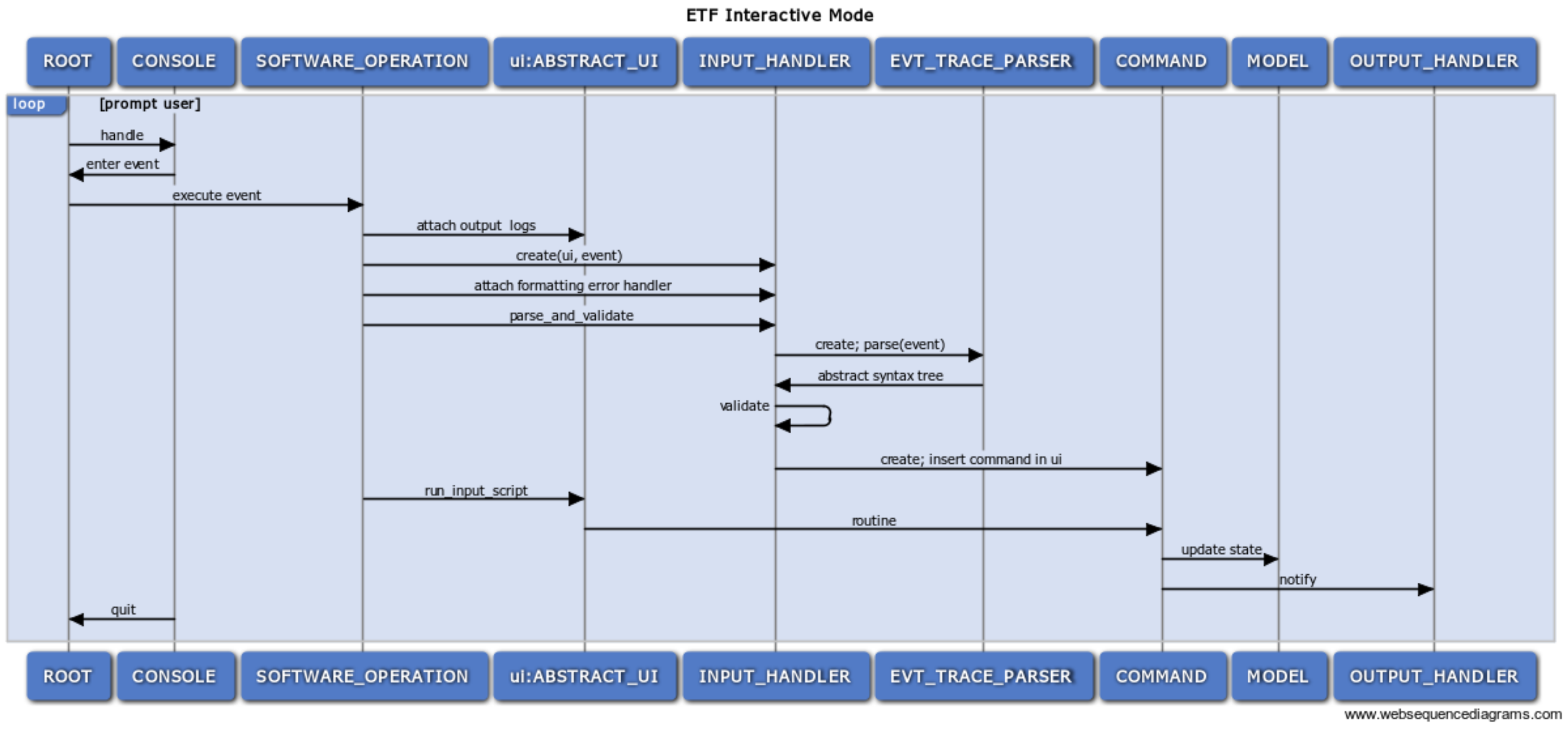


Figure 3: Running ETF in the Interactive Mode

9.7 Eiffel Code Listings of the Bank

9.7.1 Class Implementing the Singleton Pattern for Bank Access

```
note
    description: "Summary description for {BANK_ACCESS}."
    author: ""
    date: "$Date$"
    revision: "$Revision$"

expanded class
    BANK_ACCESS

feature
    bank: BANK
        once
            create Result.make
        end

invariant
    bank = bank

end
```

9.7.2 Bank as a Collection of Accounts

```
note
    description: "Summary description for {BANK}."
    author: ""
    date: "$Date$"
    revision: "$Revision$"

class
    BANK
inherit
    ANY
        redefine out end
create {BANK_ACCESS}
    make

feature {NONE} -- Initialization
    make
            -- Initialization for `Current'.
        do
            create accounts.make (10)
            accounts.compare_objects
        end

feature -- attributes

    accounts: HASH_TABLE[ACCOUNT, STRING]

    total: INTEGER

feature -- commands

    new(a_name: STRING)
        require
            not accounts.has (a_name)
        local
            l_account: ACCOUNT
        do
            create l_account.make (a_name)
            accounts.extend (l_account, a_name)
        end

    deposit(a_name:STRING; a_value:INTEGER)
        require
            accounts.has (a_name)
            accounts[a_name] /= Void
        local
            l_account: ACCOUNT
        do
            check attached accounts[a_name] as a then
                l_account := a
            end
            l_account.deposit (a_value)
            total := total + a_value
        end

    withdraw(a_name:STRING; a_value:INTEGER)
        require
                accounts.has (a_name)
            and then item(a_name).balance - a_value >= 0
        local
            l_account: ACCOUNT
        do
            check attached accounts[a_name] as a then
                l_account := a
            end
```

```
          l_account.withdraw (a_value)
          total := total − a_value
       end


    item(a_name: STRING): ACCOUNT
          −− return attached account
       require
          accounts.has (a_name)
       do
          check attached accounts[a_name] as a then
             Result := a
          end
       ensure
          Result = accounts[a_name]
       end

    out: STRING
       local
       do
          Result := " total: " + total.out + "%N"
          across accounts as a
          loop
             Result := Result + a.item.out + "%N"
          end
       end

feature −− wipe clean
    reset
       do
          accounts.wipe_out
          total := 0
       end
end
```

9.7.3 Account

```
note
    description: "Summary description for {ACCOUNT}."
    author: ""
    date: "$Date$"
    revision: "$Revision$"

class
    ACCOUNT
inherit
    ANY
        redefine out end
create
    make
feature
    make(a_name: STRING)
        do
            name := a_name
        end
feature
    balance: INTEGER
    name: STRING

    deposit(a_value:INTEGER)
        require
            a_value >= 0
        do
            balance := balance + a_value
        end

    withdraw(a_value:INTEGER)
        require
            balance - a_value >= 0
        do
            balance := balance - a_value
        end

    out: STRING
        do
            Result := " " + name + "." + "balance: " + balance.out
        end
invariant
    balance >= 0

end
```

9.8 Eiffel Code Listings of the User Commands

```
note
    description: ""
    author: ""
    date: "$Date$"
    revision: "$Revision$"

class
    ETF_NEW
inherit
    ETF_NEW_INTERFACE
create
    make
feature -- command
    new(n: STRING)
        do
            -- perform some update on the model state
            model.default_update
            etf_cmd_container.on_change.notify ([Current])
        end

end
```

```
note
    description: ""
    author: ""
    date: "$Date$"
    revision: "$Revision$"

class
    ETF_WITHDRAW
inherit
    ETF_WITHDRAW_INTERFACE
create
    make
feature -- command
    withdraw(n: STRING ; i: INTEGER)
        do
            -- perform some update on the model state
            model.default_update
            etf_cmd_container.on_change.notify ([Current])
        end

end
```