Eiffel Testing Framework (ETF): Automated Regression & Acceptance Testing

EECS3311 A & E: Software Design
Fall 2020

CHEN-WEI WANG

Learning Objectives

Upon completing this lecture, you are expected to understand:

1. ***User Interface***: Concrete vs. ***Abstract***
2. ***Use Case***: Interleaving Model, Events & ***(Abstract) States***
3. ***Acceptance Tests*** vs. Unit Tests
4. *Regression Tests*

Required Tutorial

All technical details of ETF are discussed in this tutorial series:

`https://www.youtube.com/playlist?list=PL5dxAmCmjv_5unIgLB9XiLwBey105y3kI`

Take-Home Message

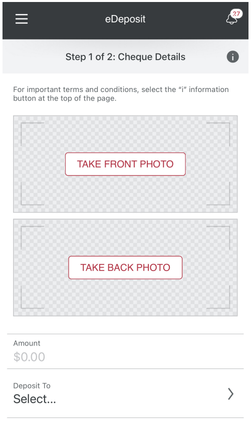
- Your remaining assignments are related to ETF: Lab3 & Project.
- You are no longer just given **partially** implemented classes:
  - Design decisions have already been made for you.
  - You are just to fill in the blanks (`to-do's`).
- ETF is in Eiffel, but try to see beyond what it allows you do:
  1. Design ***your own classes and routines***.
  2. Practice ***design principles***:
     e.g., DbC, modularity, information hiding, single-choice, cohesion.
  3. Practice ***design patterns***:
     e.g., iterator, singleton.
  4. Practice ***acceptance*** testing and ***regression*** testing.

Bank ATM: Concrete User Interfaces

An ATM app has many **concrete** (implemented, functioning) UIs.

PHYSICAL INTERFACE

MOBILE INTERFACE



Step 1 of 2: Cheque Details

For important terms and conditions, select the "i" information button at the top of the page.

Amount
$0.00

Deposit To
Select...

UI, Model, TDD

- ***Separation of Concerns***
  - The (**Concrete**) User Interface
    Users typically interact with your application via some GUI.
    e.g., web app, mobile app, or desktop app
  - The *Model* (Business Logic)
    Develop an application via classes and features.
    e.g., a bank storing, processing, retrieving accounts & transactions
- ***Test Driven Development*** (***TDD***) In practice:
  - The model should be ***independent*** of the UI or View.
  - Do **not** wait to test the *model* when the **concrete** UI is built.

⇒ Test your software as if it was a real app

way before dedicating to the design of an actual GUI.

⇒ Use an ***abstract*** UI (e.g., a cmd-line UI) for this purpose.

Prototyping System with Abstract UI

- For you to quickly *prototype* a working system, you do not need to spend time on developing a elaborate, full-fledged GUI.
- The ***Eiffel Testing Framework*** (***ETF***) allows you to:
  - Generate a starter project from the specification of an ***abstract UI***.
  - Focus on developing the business *model*.
  - Test your business model as if it were a real app.
- **Q**. What is an ***abstract UI***?
  Events ***abstracting*** observable interactions with the concrete GUI (e.g., button clicks, text entering).
- **Q**. Events vs. Features (attributes & routines)?

Events	**Features**
interactions	computations
external	internal
observable	hidden
acceptance tests	unit tests
users, customers	programmers, developers

Bank ATM: Abstract UI

Abstract UI is the list of **events** ***abstracting*** observable interactions with the concrete GUI (e.g., button clicks, text entering).

```
system bank

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

Bank ATM: Abstract States

Abstract State is a representation of the system:

- ***Including*** relevant details of functionalities under *testing*
- ***Excluding*** other irrelevant details

e.g., An ***abstract state*** may show each account's owner:

```
{alan, mark, tom}
```

e.g., An ***abstract state*** may also show each account's balance:

```
{alan: 200, mark: 300, tom: 700}
```

e.g., An ***abstract state*** may show account's transactions:

```
Account Owner: alan
  List of transactions:
    + deposit (Oct 15): $100
    - withdraw (Oct 18): $50
Account Owner: mark
  List of transactions:
```

Bank ATM: Inputs of Acceptance Tests

An *acceptance test* is a ***use case*** of the system under test, characterized by sequential occurrences of ***abstract events***.

For example:

```
new("alan")
new("mark")
deposit("alan", 200)
deposit("mark", 100)
transfer("alan", "mark", 50)
```

Bank ATM: Outputs of Acceptance Tests (1)

Output from running an ***acceptance test*** is a sequence interleaving ***abstract states*** and ***abstract events***:

$$S_0 -> e_1 -> S_1 -> e_2 -> S_2 -> \ldots$$

where:

- S_0 is the ***initial state***.
- S_i is the ***pre-state*** of event e_{i+1} [$i \geq 0$]
  e.g., S_0 is the pre-state of e_1, S_1 is the pre-state of e_2
- S_i is the ***post-state*** of event e_i [$i \geq 1$]
  e.g., S_1 is the post-state of e_1, S_2 is the post-state of e_2

Bank ATM: Outputs of Acceptance Tests (2)

Consider an example acceptance test output:

```
  {}
->new("alan")
  {alan: 0}
->new("mark")
  {alan: 0, mark: 0}
->deposit("alan", 200)
  {alan: 200, mark: 0}
->deposit("mark", 100)
  {alan: 200, mark: 100}
->transfer("alan", "mark", 50)
  {alan: 150, mark: 150}
```

- ***Initial State***? `{}`
- What role does the state `{alan: 200, mark: 0}` play?
  - ***Post-State*** of `deposit("alan", 200)`
  - ***Pre-State*** of `deposit("mark", 100)`

Bank ATM: Acceptance Tests vs. Unit Tests

Q. Difference between an ***acceptance test*** and a ***unit test***?

```
  {}
->new("alan")
  {alan: 0}
->deposit("alan", 200)
  {alan: 200}
```

```
test: BOOLEAN
  local acc: ACCOUNT
  do create acc.make("alan")
     acc.add(200)
     Result := acc.balance = 200
  end
```

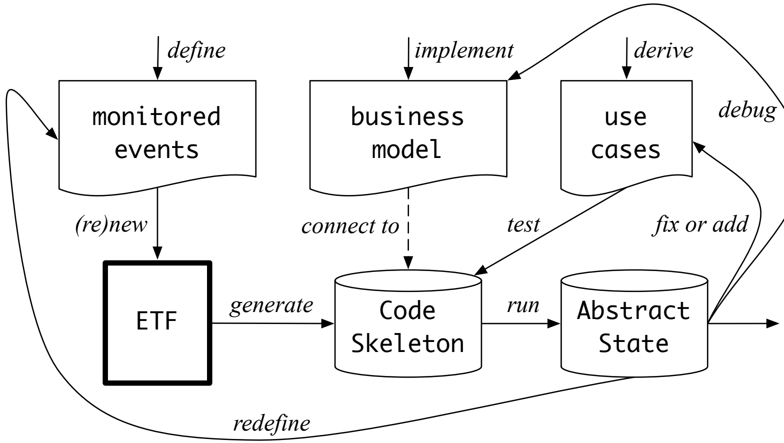
A.

- Writing a ***unit test*** requires knowledge about the ***programming language*** and details of ***implementation***.
  $\Rightarrow$ Written and run by developers
- Writing an ***acceptance test*** only requires familiarity with the ***abstract UI*** and ***abstract state***.
  $\Rightarrow$ Written and run by customers [for communication]
  $\Rightarrow$ Written and run by developers [for testing]

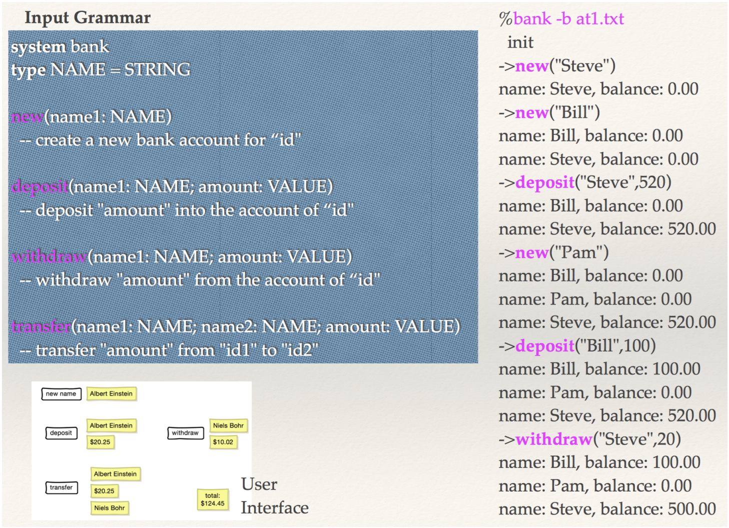
ETF in a Nutshell

- **Eiffel Testing Framework** (**ETF**) facilitates engineers to write and execute ***input-output-based acceptance tests***.
  - ***Inputs*** are specified as traces of events (or sequences).
  - The ***abstract UI*** of the system under development (SUD) is defined by declaring the list of input events that might occur.
  - ***Outputs*** are interleaved states and events logged to the terminal, and their formats may be customized.
- An *executable* ETF project tailored for the SUD can already be generated, using these ***event declarations*** (specified in a plain text file), with a default *business model*.
  - Once the *business model* is implemented, there is a small number of steps to follow for developers to connect it to the generated ETF.
  - Once connected, developers may **re-run** all ***acceptance tests*** and observe if the expected state effects occur.

Workflow: Develop-Connect-Test



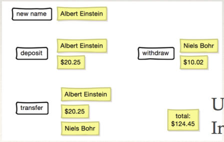
ETF: Abstract UI and Acceptance Test



Input Grammar

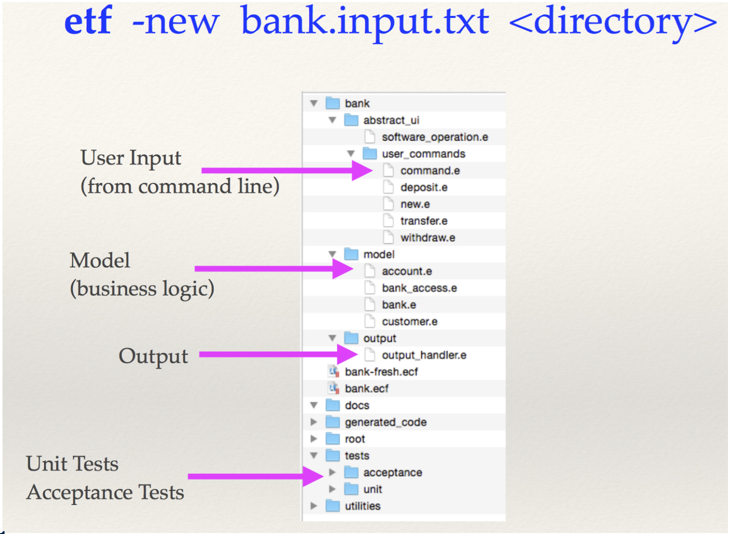
```
system bank
type NAME = STRING

new(name1: NAME)
  -- create a new bank account for “id"

deposit(name1: NAME; amount: VALUE)
  -- deposit "amount" into the account of “id"

withdraw(name1: NAME; amount: VALUE)
  -- withdraw "amount" from the account of “id"

transfer(name1: NAME; name2: NAME; amount: VALUE)
  -- transfer "amount" from "id1" to "id2"
```

```
%bank -b at1.txt
 init
->new("Steve")
name: Steve, balance: 0.00
->new("Bill")
name: Bill, balance: 0.00
name: Steve, balance: 0.00
->deposit("Steve",520)
name: Bill, balance: 0.00
name: Steve, balance: 520.00
->new("Pam")
name: Bill, balance: 0.00
name: Pam, balance: 0.00
name: Steve, balance: 520.00
->deposit("Bill",100)
name: Bill, balance: 100.00
name: Pam, balance: 0.00
name: Steve, balance: 520.00
->withdraw("Steve",20)
name: Bill, balance: 100.00
name: Pam, balance: 0.00
name: Steve, balance: 500.00
```



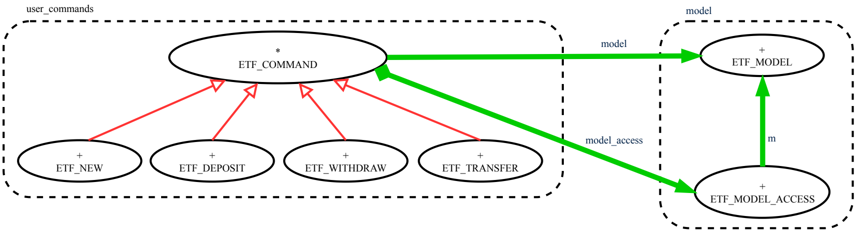
User Interface

ETF: Generating a New Project


```
etf -new bank.input.txt <directory>
```

```
▼ bank
    ▼ abstract_ui
        software_operation.e
        ▼ user_commands
            command.e
            deposit.e
            new.e
            transfer.e
            withdraw.e
    ▼ model
        account.e
        bank_access.e
        bank.e
        customer.e
    ▼ output
        output_handler.e
    bank-fresh.ecf
    bank.ecf
▼ docs
▶ generated_code
▶ root
▼ tests
    ▶ acceptance
    ▶ unit
▶ utilities
```

ETF: Architecture



- Classes in the `model` cluster are hidden from the users.
- All commands reference to the same model (bank) instance.
- When a user's request is made:
  - A ***command object*** of the corresponding type is created, which invokes relevant feature(s) in the `model` cluster.
  - Updates to the model are published to the output handler.

ETF: Implementing an Abstract Command

```
class
  ETF_DEPOSIT
inherit
  ETF_DEPOSIT_INTERFACE
    redefine deposit end
create
  make
feature -- command
  deposit(id: STRING ; amount: REAL_64)
    do
      if not model.has_user (id) then
        -- Set some error message
      elseif not amount <= model.get_balance (id)  then
        -- Set some other error message
      else
        -- perform some update on the model state
        model.deposit (id, amount)
      end
      -- Publish model update
      etf_cmd_container.on_change.notify ([Current])
    end
end
```

Beyond this lecture

The ***singleton*** pattern is instantiated in the ETF framework:

- `ETF_MODEL` (***shared data***)
- `ETF_MODEL_ACCESS` (***exclusive once access***)
- `ETF_COMMAND` and its effective descendants:

```
deferred class
  ETF_COMMAND
feature -- Attributes
  model: ETF_MODEL
feature {NONE}
  make(...)
    local
      ma: ETF_MODEL_ACCESS
    do
      ...
      model := ma.m
    end
end
```

```
class
  ETF_DEPOSIT
inherit
  ETF_DEPOSIT_INTERFACE
  -- which inherits ETF_COMMAND
feature -- command
  deposit(...)
    do
      ...
      model.some_routine (...)
      ...
    end
end
```

Index (1)

Index (2)