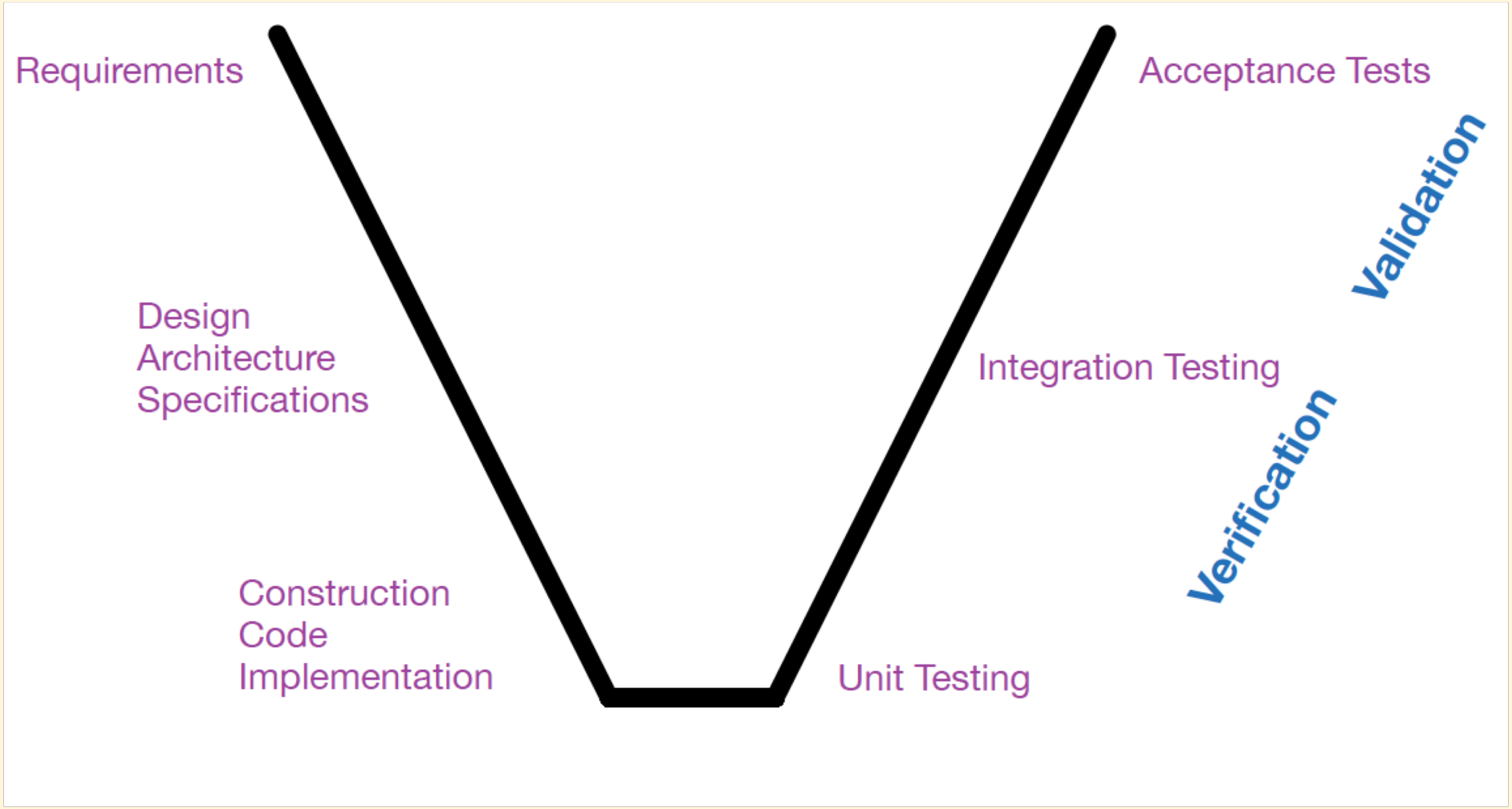
# Eiffel Testing Framework
# ETF

Decouple model (the main business logic)
from the user interface (view/controller)


Requirements
Acceptance Tests
Validation
Design
Architecture
Specifications
Integration Testing
Verification
Construction
Code
Implementation
Unit Testing

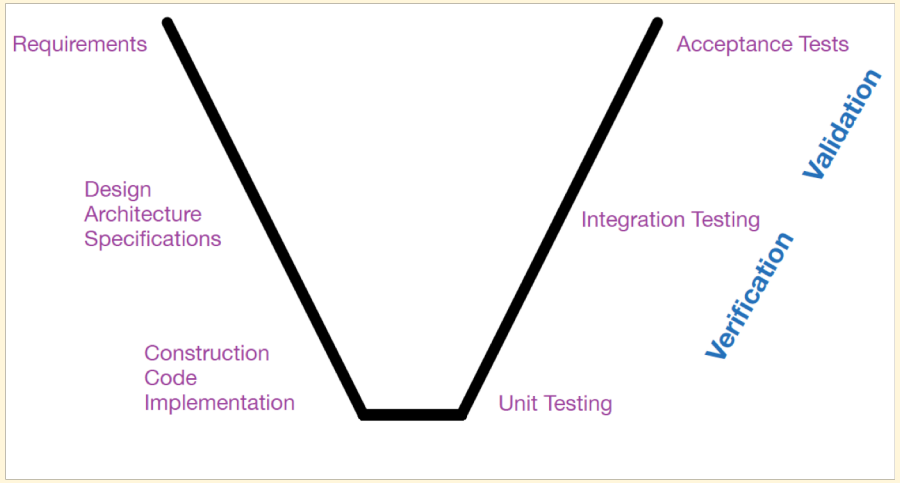
The terms **Verification** and **Validation** are commonly used in software engineering to mean two different types of analysis. The usual definitions are:

**Validation**: Are we building the right system?

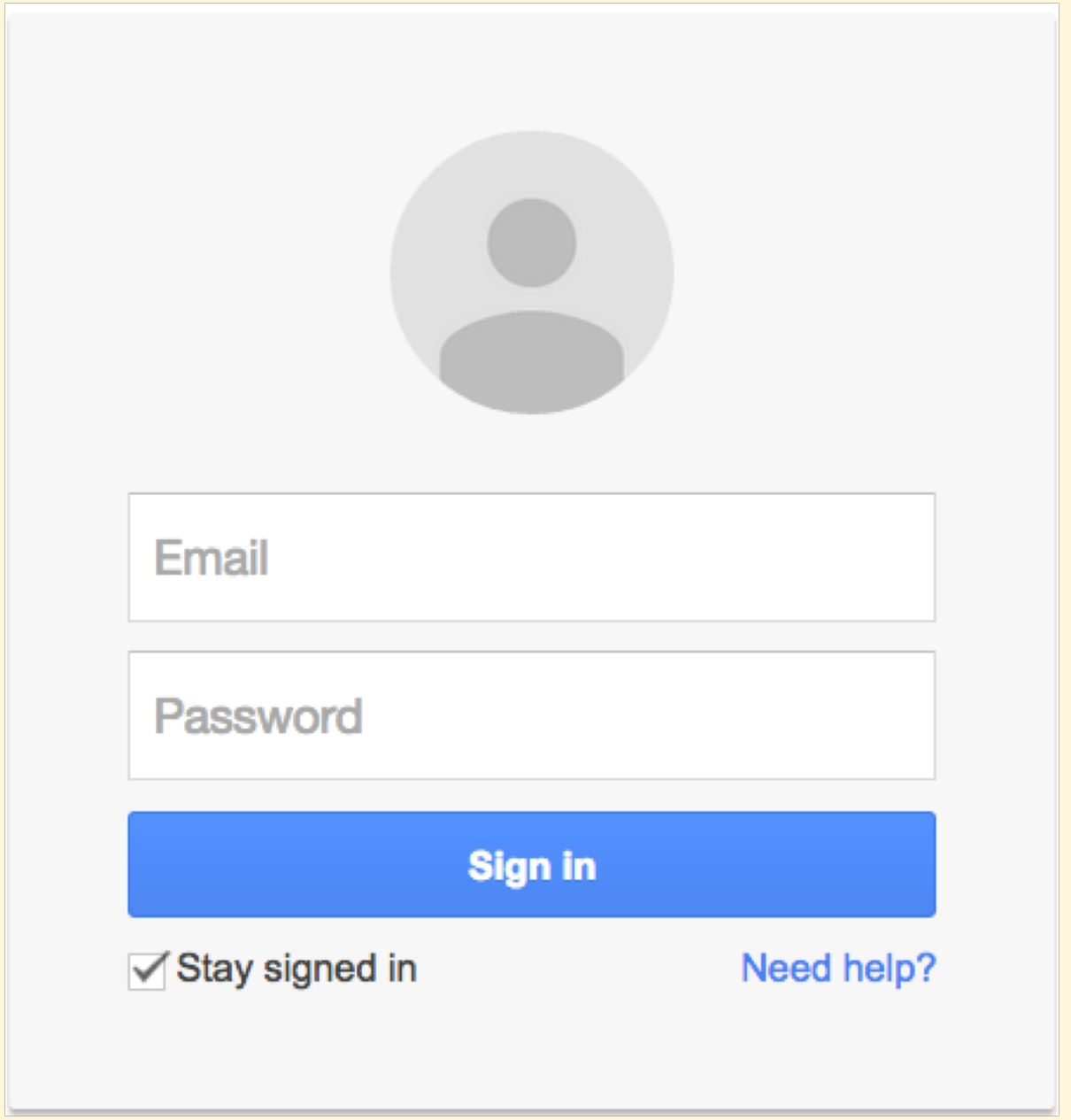
**Verification**: Are we building the system right?

**Validation** is concerned with checking that the system will meet the customer's actual needs (the requirements), while **Verification** is concerned with whether the system is well-engineered, safe, and error-free.

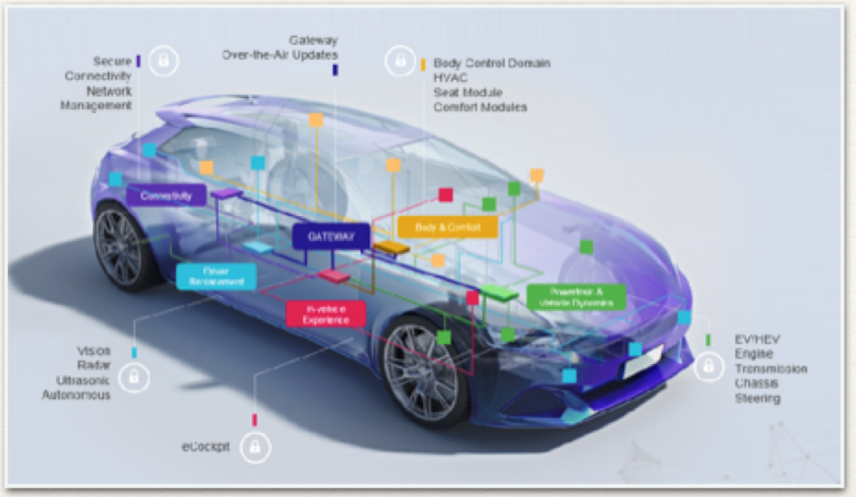
Verification will help to determine whether the software is of high quality, but it will not ensure that the system is fit for use and useful to the customer.



# User Interface




por Sergio Ruiz / Flickr / CC




https://blog.nxp.com/automotive/cars-are-made-of-code

**Behind each system is a a program running the business logic that encodes the real-world business rules, data and algorithms**

**(slide from Li Yi)**

news.sky.com

Yelda Ozkoca / More Than Shipping

# Bank Requirements

The banking system shall allows Tellers at the bank to add new accounts by id. Any customer can deposit money into an account specified by id, withdraw money, and transfer money from an account with one id to an account with another id. Tellers check that customers have access to the relevant accounts. Eventually ATMs will take over the management of these services, but for now it is Tellers who do the transactions on behalf of the bank customers. Balances in accounts must be non-negative. The system shall keep track of the total money that should be in the bank independently of amounts in individual accounts.

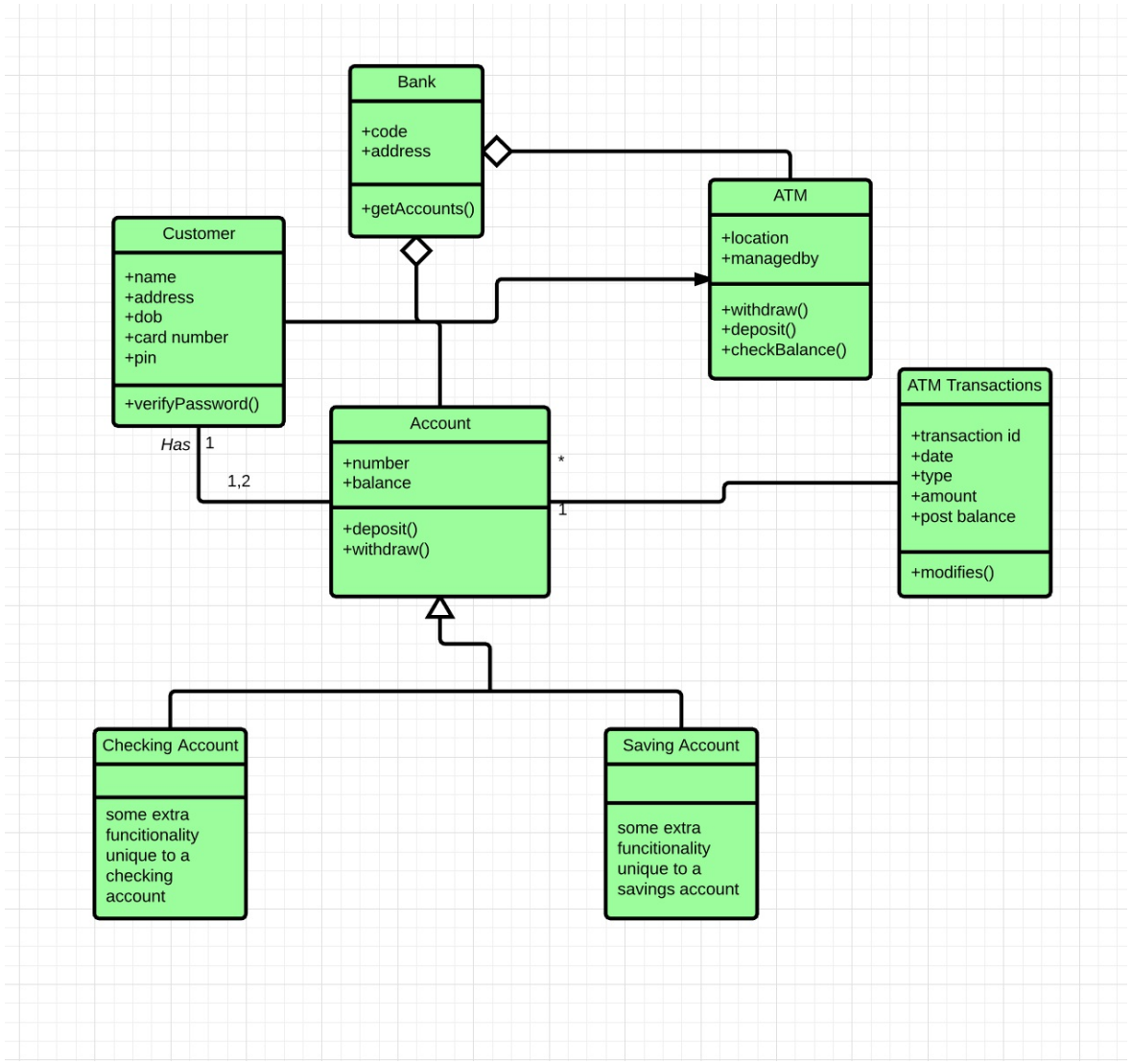
# Concrete User Interface?


- What about *transfer*(id1,id2, amount)?
- Is it not premature to get into the design of the concrete user interface?
- How about describing the UI abstractly

Actor

**User interface**

## Business Logic (the model)

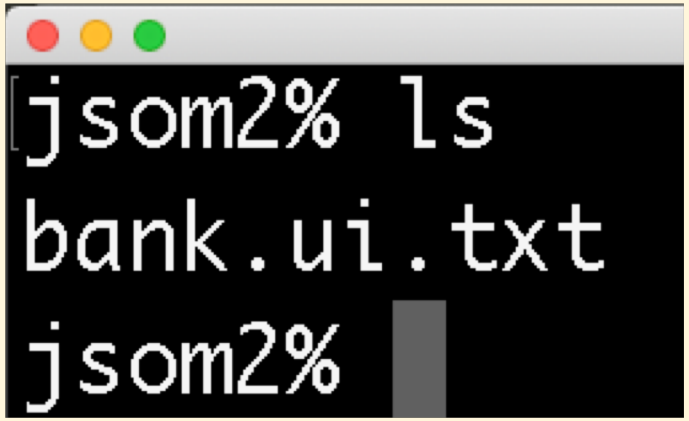


# Abstract Grammar for the User Interface

bank.ui.txt

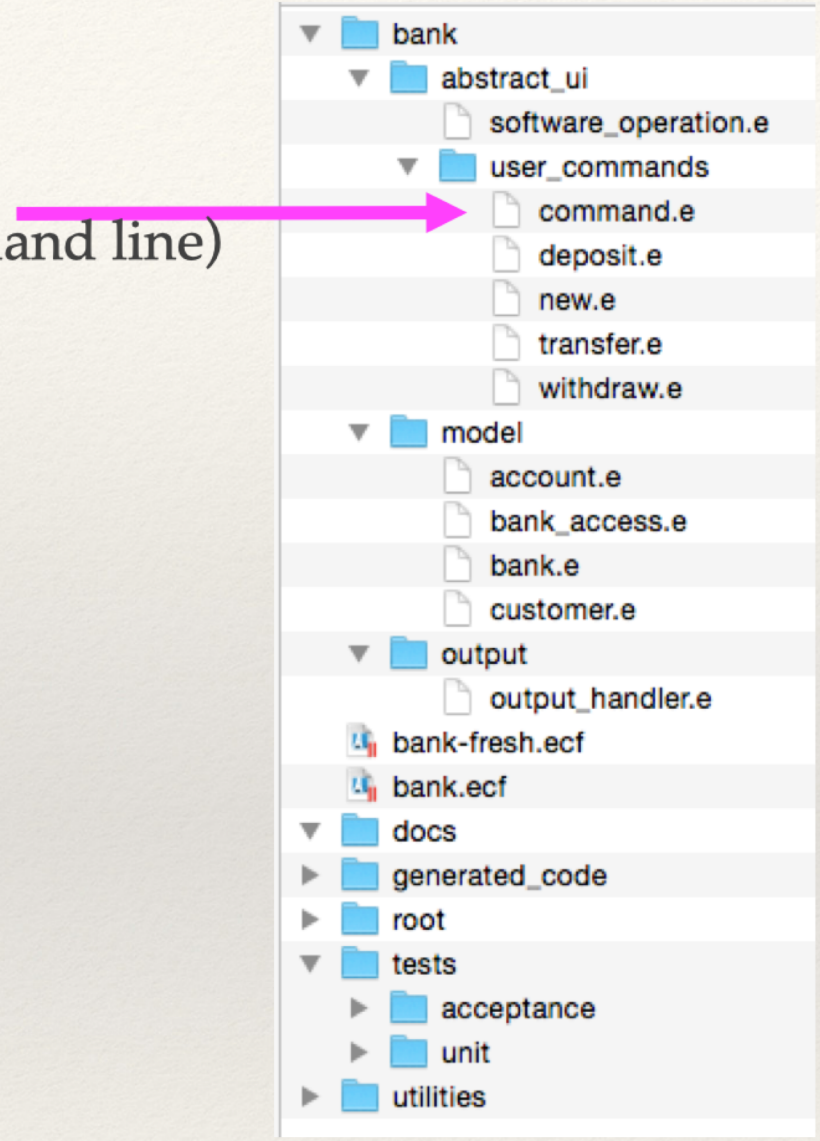
```
1  system bank
2  -- small bank
3
4  type ID = STRING
5  new(a_id: ID) -- add a new account id to bank
6
7  deposit(a_id: ID; a_value: VALUE)
8  withdraw(a_id:ID; a_value: VALUE)
9  transfer(id1: ID; id2: ID; a_value: VALUE)
10
```

# Create an ETF project



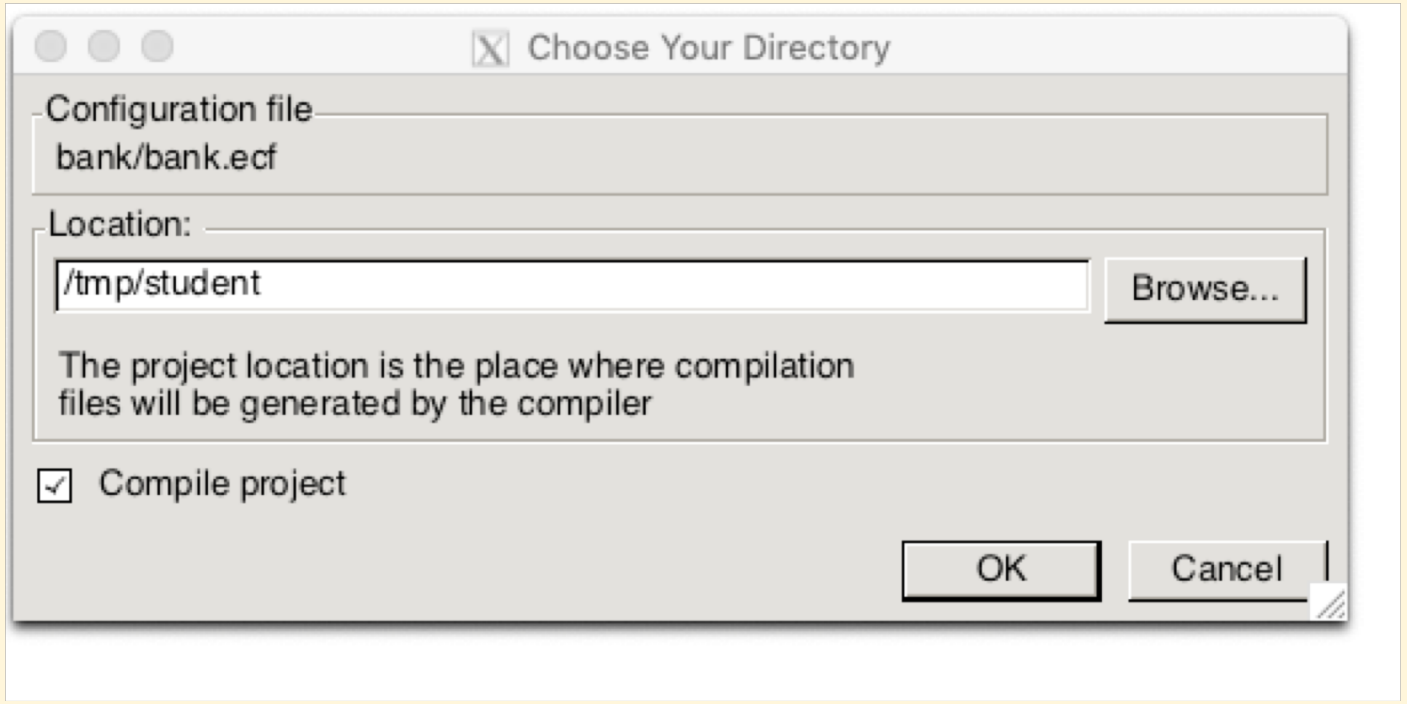

```
jsom2% ls
bank.ui.txt
jsom2% mkdir bank
jsom2% etf -new bank.ui.txt bank
File created    : bank/bank-fresh.ecf
File created    : bank/bank.ecf
```

User Input
(from command line)

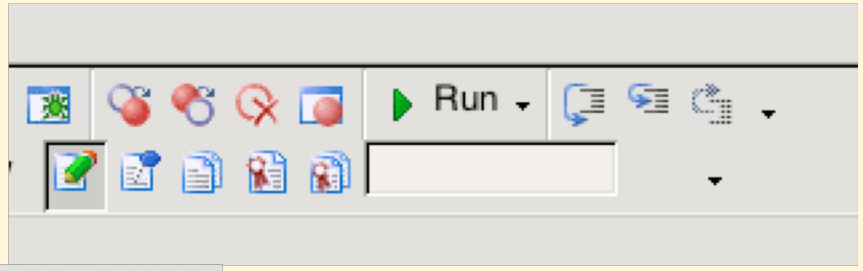


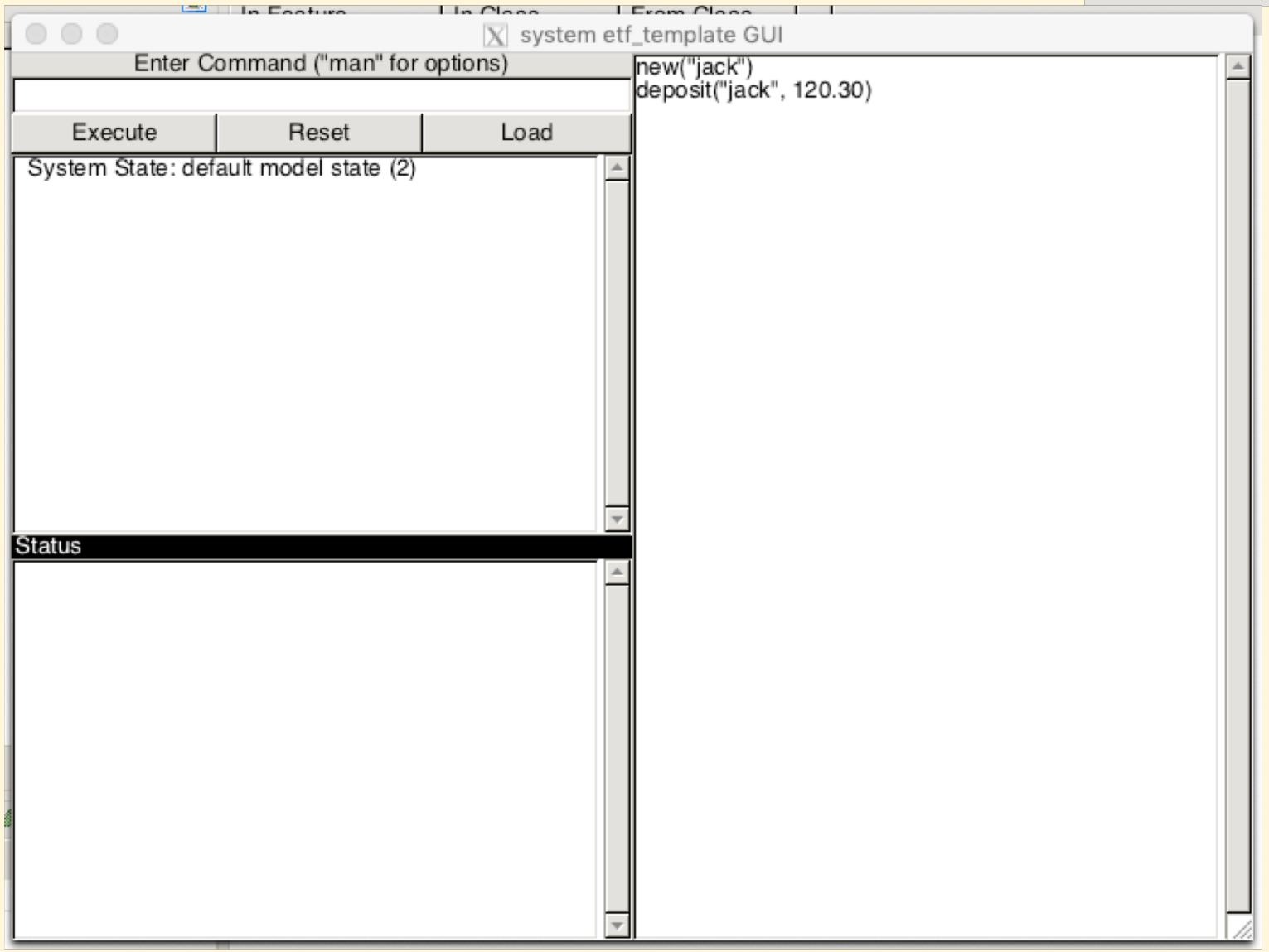
# Launch estudio

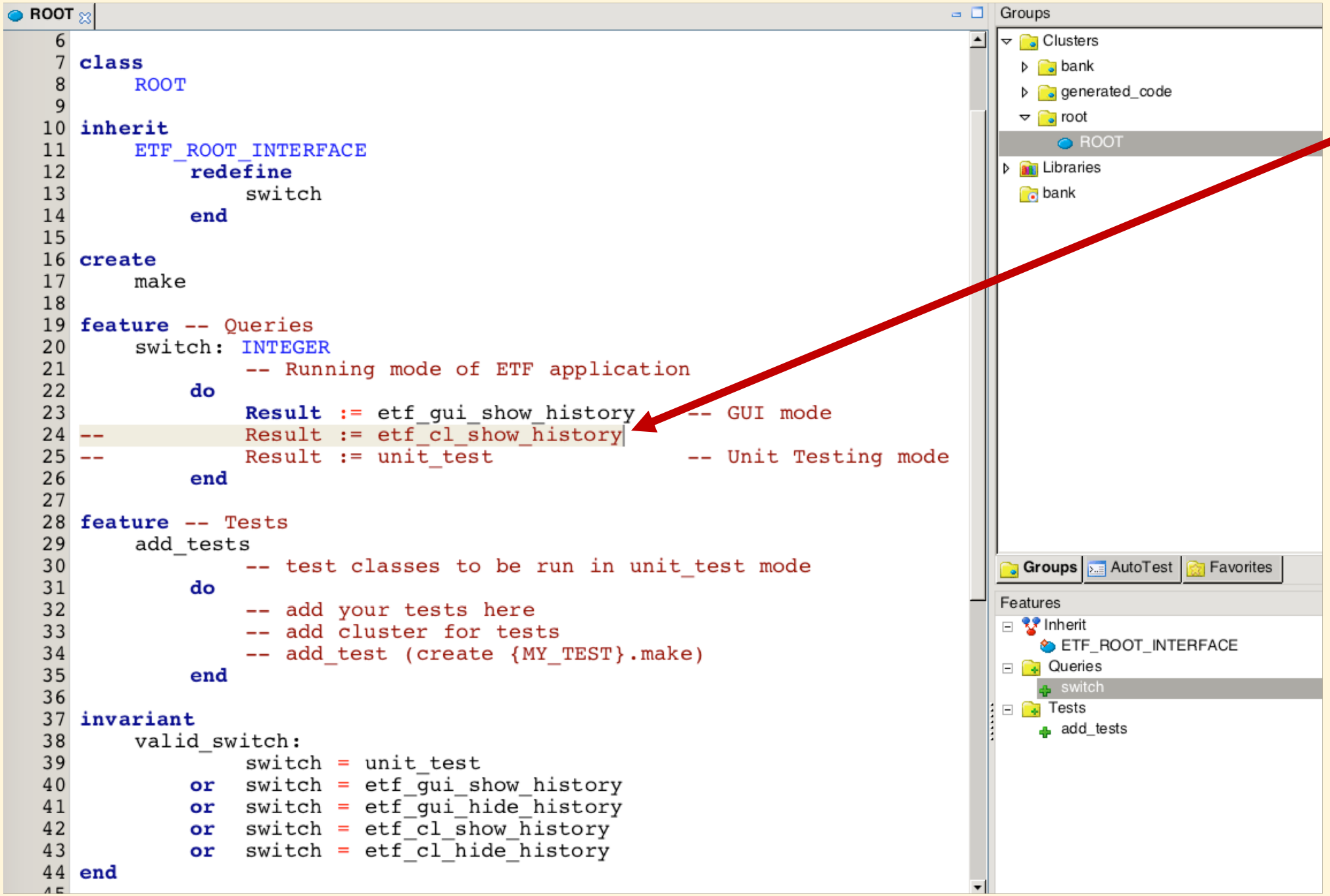
```
jsom2% estudio bank/bank.ecf &
[1] 1465
```



# Run

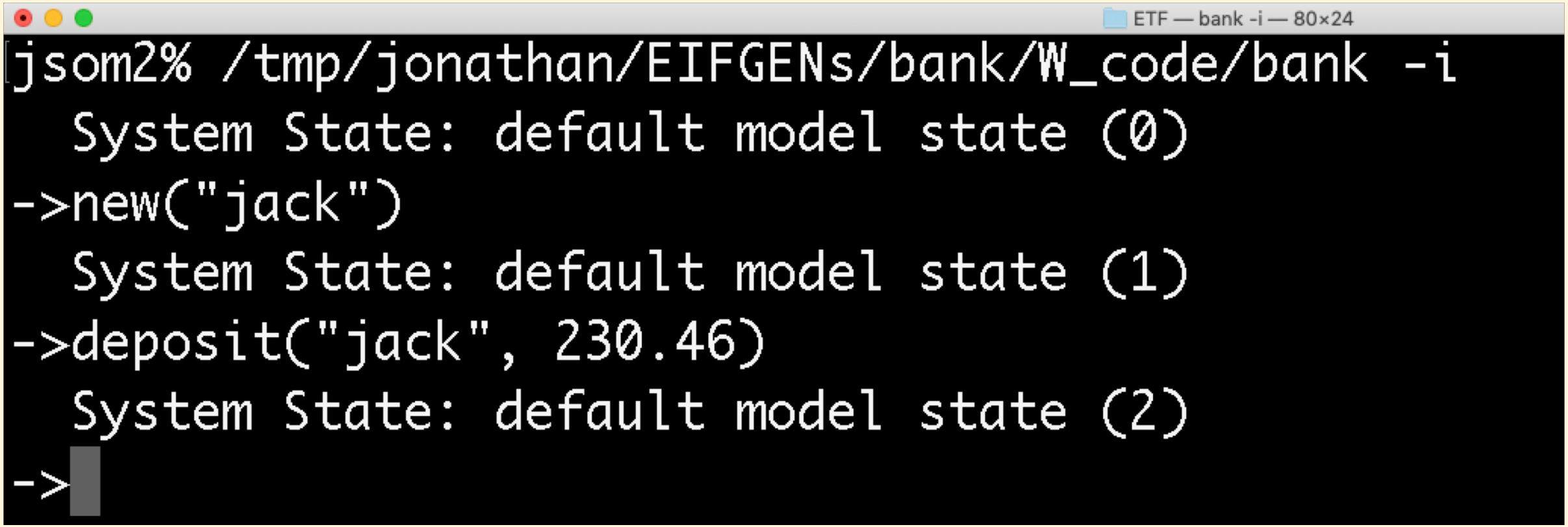

Run


system etf_template GUI
Enter Command ("man" for options)
Execute
Reset
Load
System State: default model state (2)
Status
new("jack")
deposit("jack", 120.30)

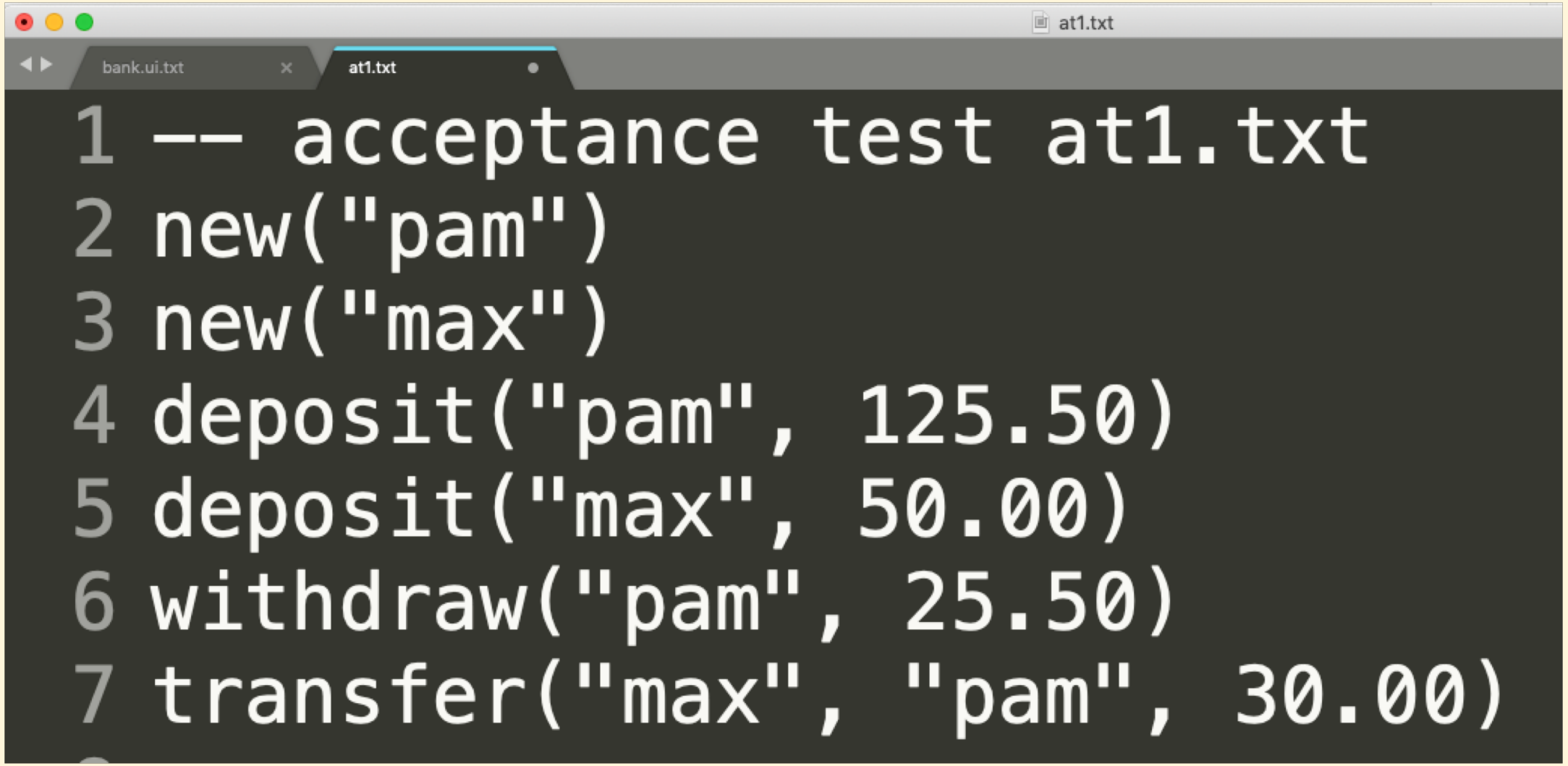

ROOT
class
ROOT
inherit
ETF_ROOT_INTERFACE
redefine
switch
end
create
make
feature -- Queries
switch: INTEGER
-- Running mode of ETF application
do
Result := etf_gui_show_history -- GUI mode
-- Result := etf_cl_show_history
-- Result := unit_test -- Unit Testing mode
end
feature -- Tests
add_tests
-- test classes to be run in unit_test mode
do
-- add your tests here
-- add cluster for tests
-- add_test (create {MY_TEST}.make)
end
invariant
valid_switch:
switch = unit_test
or switch = etf_gui_show_history
or switch = etf_gui_hide_history
or switch = etf_cl_show_history
or switch = etf_cl_hide_history
end
Groups
Clusters
bank
generated_code
root
ROOT
Libraries
bank
Groups
AutoTest
Favorites
Features
Inherit
ETF_ROOT_INTERFACE
Queries
switch
Tests
add_tests

Uncomment

# Interactive mode



# Write an acceptance test at1.tx



```
red% /tmp/student/EIFGENs/bank/W_code/bank -b at1.txt
  System State: default model state (0)
->new("pam")
  System State: default model state (1)
->new("max")
  System State: default model state (2)
->deposit("pam",125.5)
  System State: default model state (3)
->deposit("max",50)
  System State: default model state (4)
->withdraw("pam",25.5)
  System State: default model state (5)
->transfer("max","pam",30)
  System State: default model state (6)
```

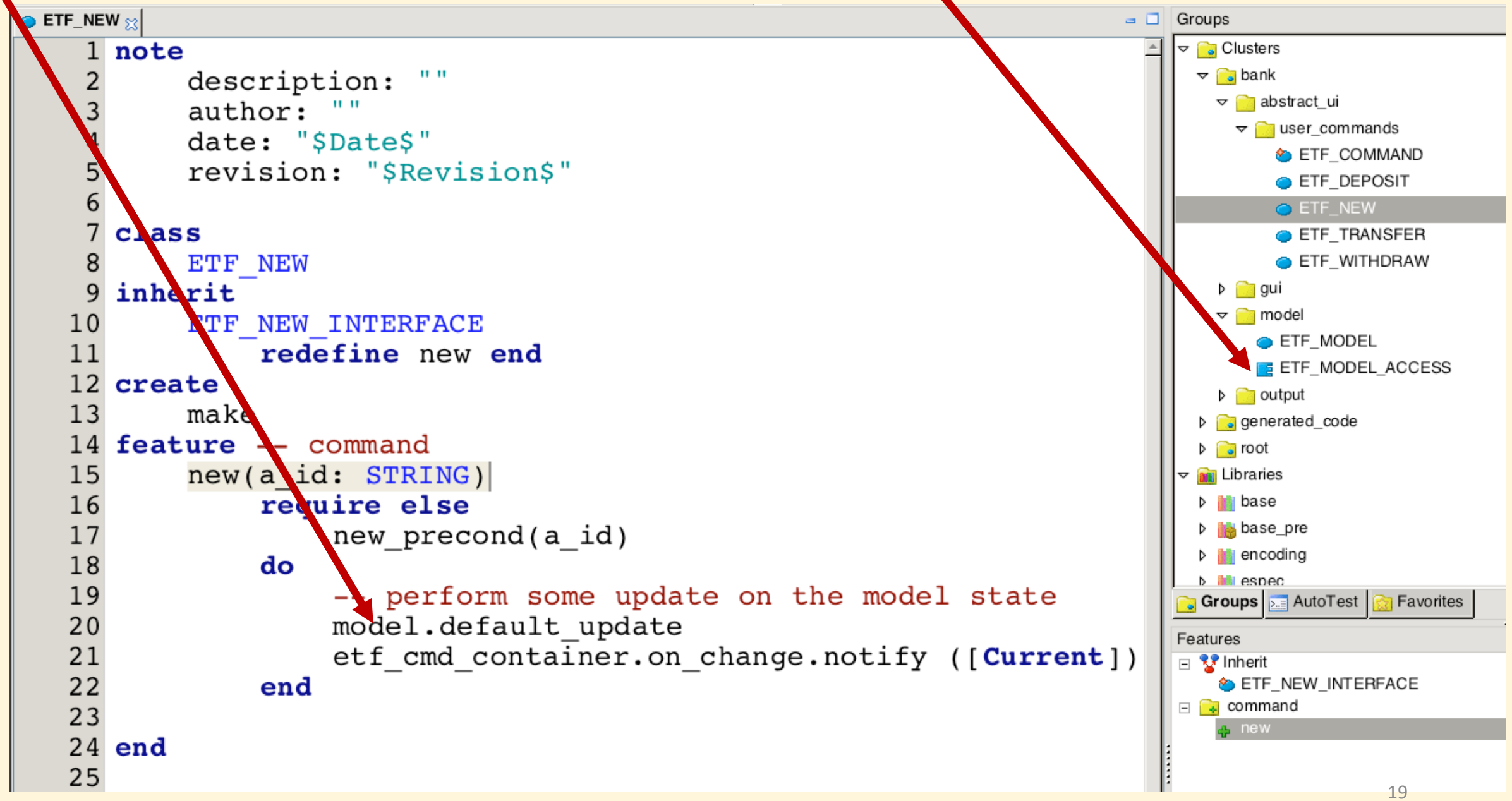
# Acceptance Test/Use Case

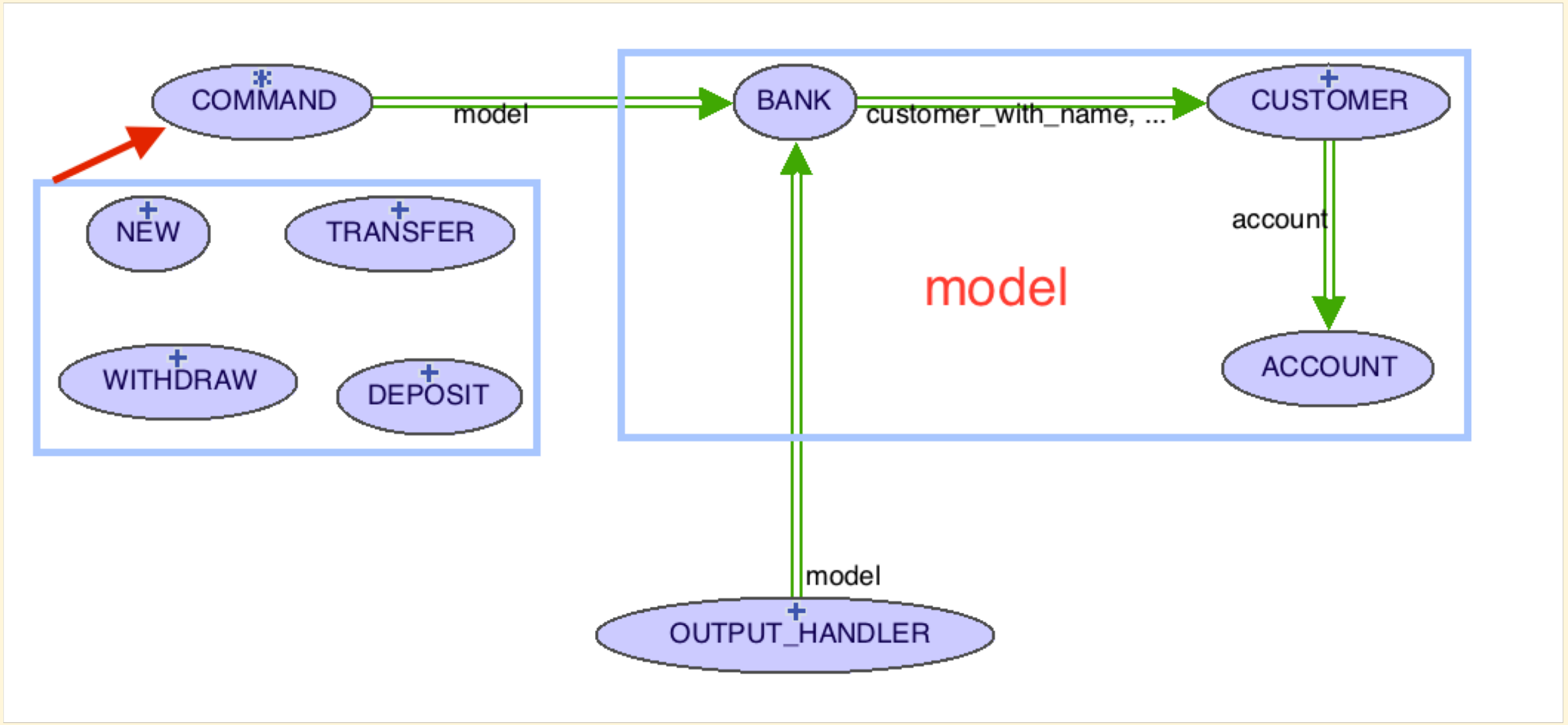
A use case is a list of actions or user steps defining the interactions between an Actor (external to the system) and a system to achieve a useful goal for an Actor. An Actor can be a human or other external system.

The Actor is your Customer

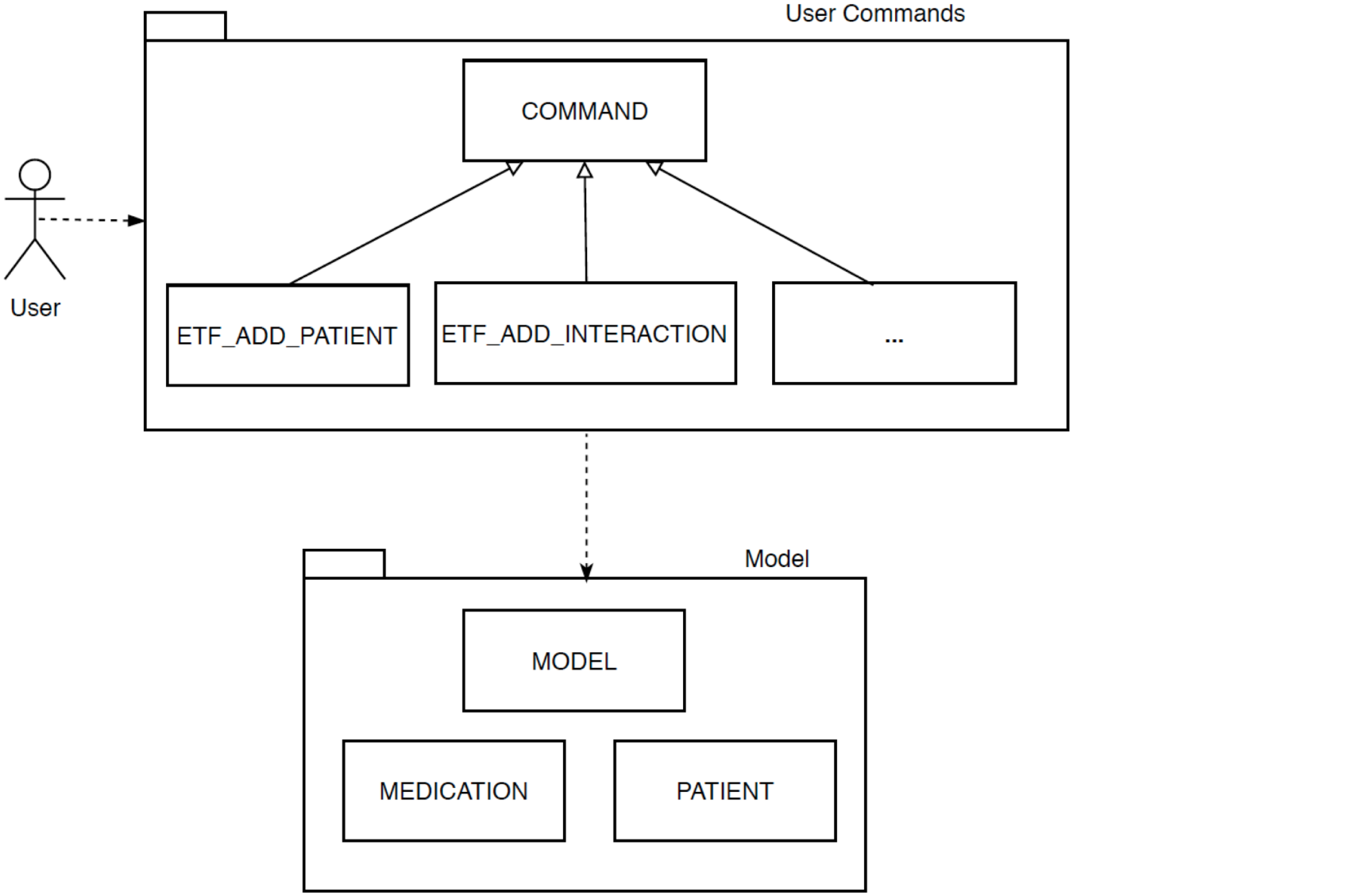
model

Singleton

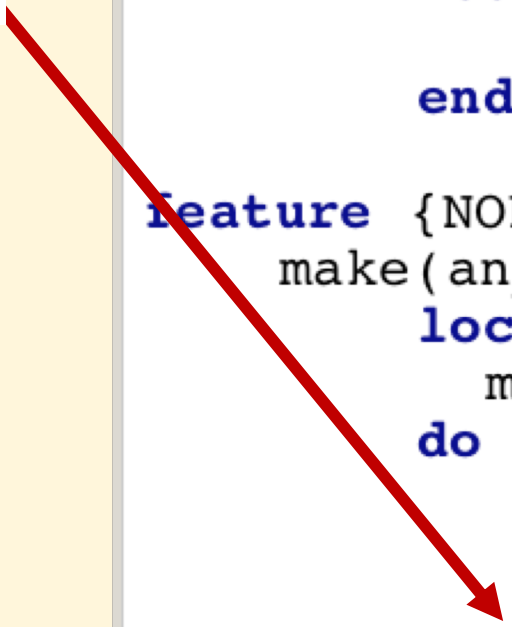



COMMAND
model
BANK
customer_with_name, ...
CUSTOMER
account
NEW
TRANSFER
WITHDRAW
DEPOSIT
model
ACCOUNT
model
OUTPUT_HANDLER

# Class Diagram: User Commands package depends on Model package



Singleton



```
deferred class
    ETF_COMMAND

inherit
    ETF_COMMAND_INTERFACE
        redefine
            make
        end

feature {NONE}
    make(an_etf_cmd_name: STRING; etf_cmd_args:
        local
          model_access: ETF_MODEL_ACCESS
        do
            Precursor(an_etf_cmd_name, etf_cmd_a
            -- may set your own model state her
            model := model_access.m
        end
```

```
class
    ETF_MODEL

inherit
    ANY
        redefine
            out
        end

create {ETF_MODEL}
    make

feature {NONE} --
    make
            -- In
        do
            creat
            i :=
        end

feature -- model attributes
    s : STRING
    i : INTEGER

feature -- model operations
    default_update
            -- Perform update to the model state.
        do
            i := i + 1
        end

    reset
            -- Reset model state.
        do
            make
        end

feature -- queries
    out : STRING
        do
            create Result.make_from_string ("   ")
            Result.append ("System State: default model state ")
            Result.append ("(")
            Result.append (i.out)
            Result.append (")")
        end

end
```

```
feature -- queries
     out : STRING
          do
               create Result.make_from_string ("   ")
               Result.append ("System State: default model state ")
               Result.append ("(")
               Result.append (i.out)
               Result.append (")")
          end
```

# class MODEL …

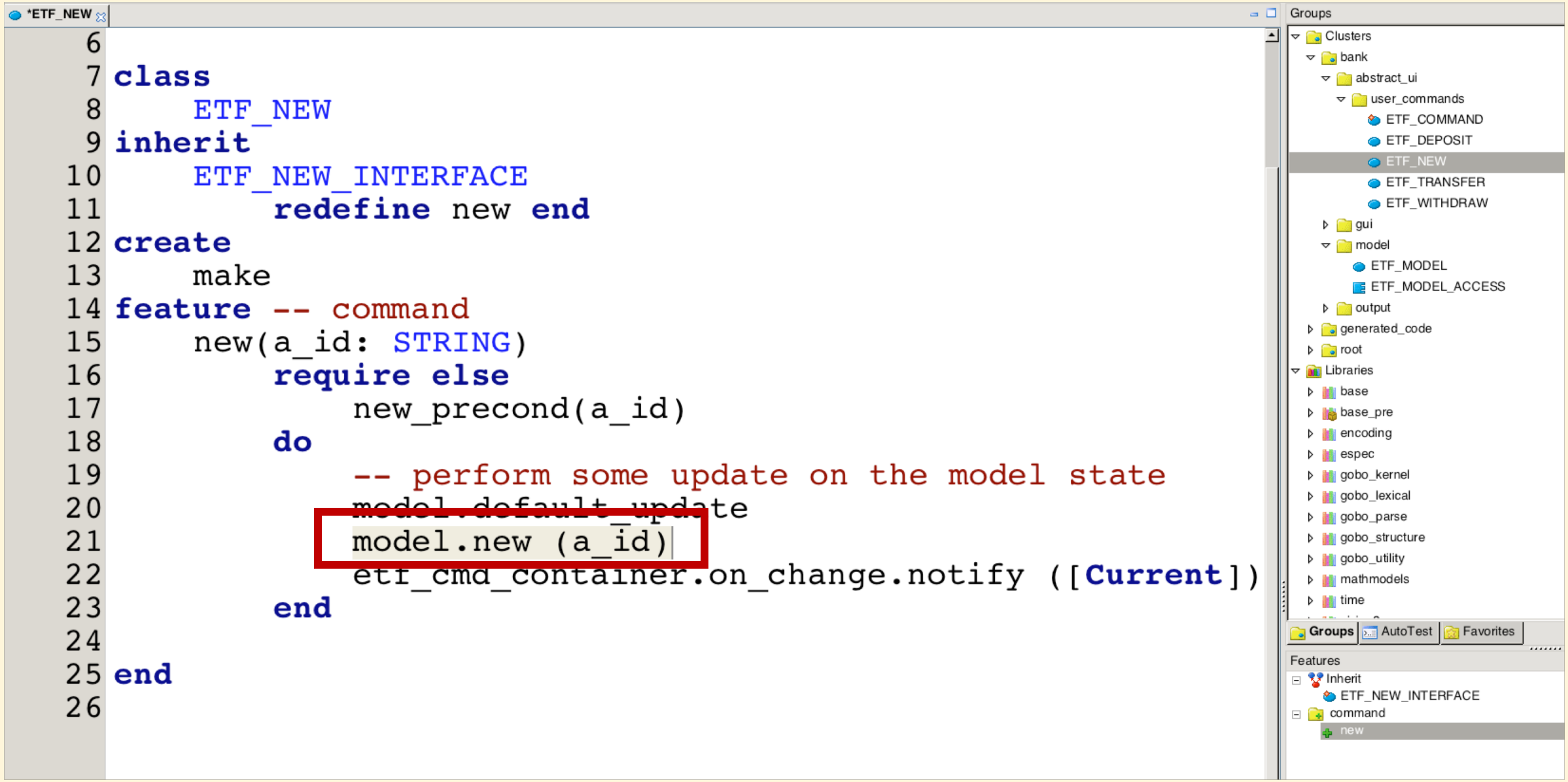
```
feature -- bank
    balance: VALUE
    id: STRING

    new(a_id: STRING)
        do
            id := a_id
        end
```

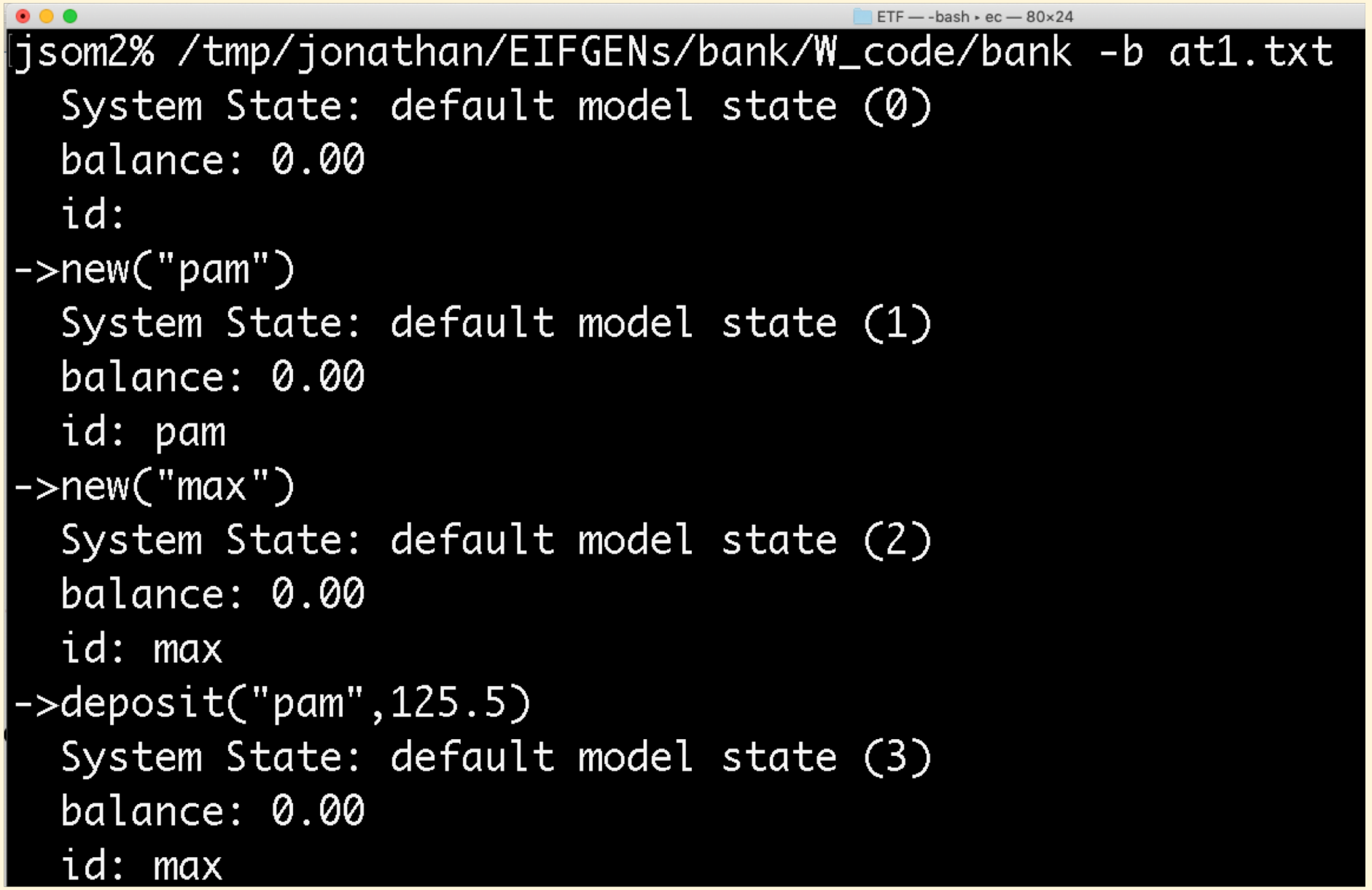
```
out : STRING
    do
        create Result.make_from_string ("  ")
        Result.append ("System State: default model state ")
        Result.append ("(")
        Result.append (i.out)
        Result.append (")")
        Result.append ("%N  balance: " + balance.out)
        Result.append ("%N  id: " + id)
    end
```

# Why does new("pam") not do anything?

```
ETF — -bash ▸ ec — 80×24
jsom2% /tmp/jonathan/EIFGENs/bank/W_code/bank -b at1.txt
   System State: default model state (0)
   balance: 0.00
   id:
->new("pam")
   System State: default model state (1)
   balance: 0.00
   id:
->new("max")
   System State: default model state (2)
   balance: 0.00
   id:
```


*ETF_NEW
6
7 class
8 ETF_NEW
9 inherit
10 ETF_NEW_INTERFACE
11 redefine new end
12 create
13 make
14 feature -- command
15 new(a_id: STRING)
16 require else
17 new_precond(a_id)
18 do
19 -- perform some update on the model state
20 model.default_update
21 model.new (a_id)
22 etf_cmd_container.on_change.notify ([Current])
23 end
24
25 end
26
Groups
Clusters
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
root
Libraries
base
base_pre
encoding
espec
gobo_kernel
gobo_lexical
gobo_parse
gobo_structure
gobo_utility
mathmodels
time
Groups
AutoTest
Favorites
Features
Inherit
ETF_NEW_INTERFACE
command
new

# Why does Pam's deposit now show?



# class MODEL …

```
feature -- bank
    balance: VALUE
    id: STRING

    new(a_id: STRING)
        do
            id := a_id
        end

    deposit(a_id: STRING ; a_value: VALUE)
        do
            id := a_id
            balance := balance + a_value
        end
```

```
class ETF_DEPOSIT feature
   deposit(a_id: STRING ; a_value: VALUE)
      do model.deposit (a_id, a_value) … end
```

```
jsom2% /tmp/jonathan/EIFGENs/bank/W_code/bank -b at2.txt
  System State: default model state (0)
  balance: 0.00
  id:
->new("pam")
  System State: default model state (1)
  balance: 0.00
  id: pam
->deposit("pam",125.5)
  System State: default model state (2)
  balance: 125.50
  id: pam
->deposit("max",50)
  System State: default model state (3)
  balance: 175.50
  id: max
```

# Design Problems

- We are using classes STRING and VALUE
  - We need classes such as ID, ACCOUNT and BANK
- We need a way to write the expected output so that we can compare it with the actual output
  - Abstract state
  - Regression testing
- Contracts are missing. Cannot withdraw too much

at1.expected.txt

at2.txt | at1.expected.txt

```
 1   System State: default model state (0)
 2   accounts:
 3 ->new("pam")
 4   System State: default model state (1)
 5   accounts:
 6     pam -> 0
 7 ->new("max")
 8   System State: default model state (2)
 9   accounts:
10     pam -> 0
11     max -> 0
12 ->deposit("pam",125.5)
13   System State: default model state (3)
14   accounts:
15     pam -> 125.50
16     max -> 0
17 ->deposit("max",50)
18   System State: default model state (4)
19   accounts:
20     pam -> 125.50
21     max -> 50
```

# at1.expected.txt

## Abstract state

account: FUN [ID, VALUE]

```
->deposit("max",50)
   accounts:
      pam -> 125.50
      max -> 50.00
-> transfer("max", "pam", 25.00)
   accounts:
      pam -> 150.50
      max -> 25.00
```

at1.expected.txt

transfer?

Abstract state

account: FUN [ID, VALUE]

# Contracts

Not good design

```
deposit(a_id: STRING ; a_value: VALUE)
    require
        -- a_id ∈ account.domain
        a_value > "0.0"
    do
        id := a_id
        balance := balance + a_value
    end
```

# Demanding Contract in Model

```
deposit(a_id: STRING ; a_value: VALUE)
    require
        -- a_id ∈ account.domain
        a_value > "0.0"
    do
        id := a_id
        balance := balance + a_value
    end
```

# Tolerant Contract at the User Interface

```
class ETF_DEPOSIT feature
   deposit(a_id: STRING ; a_value: VALUE)
      do model.deposit (a_id, a_value) … end
```

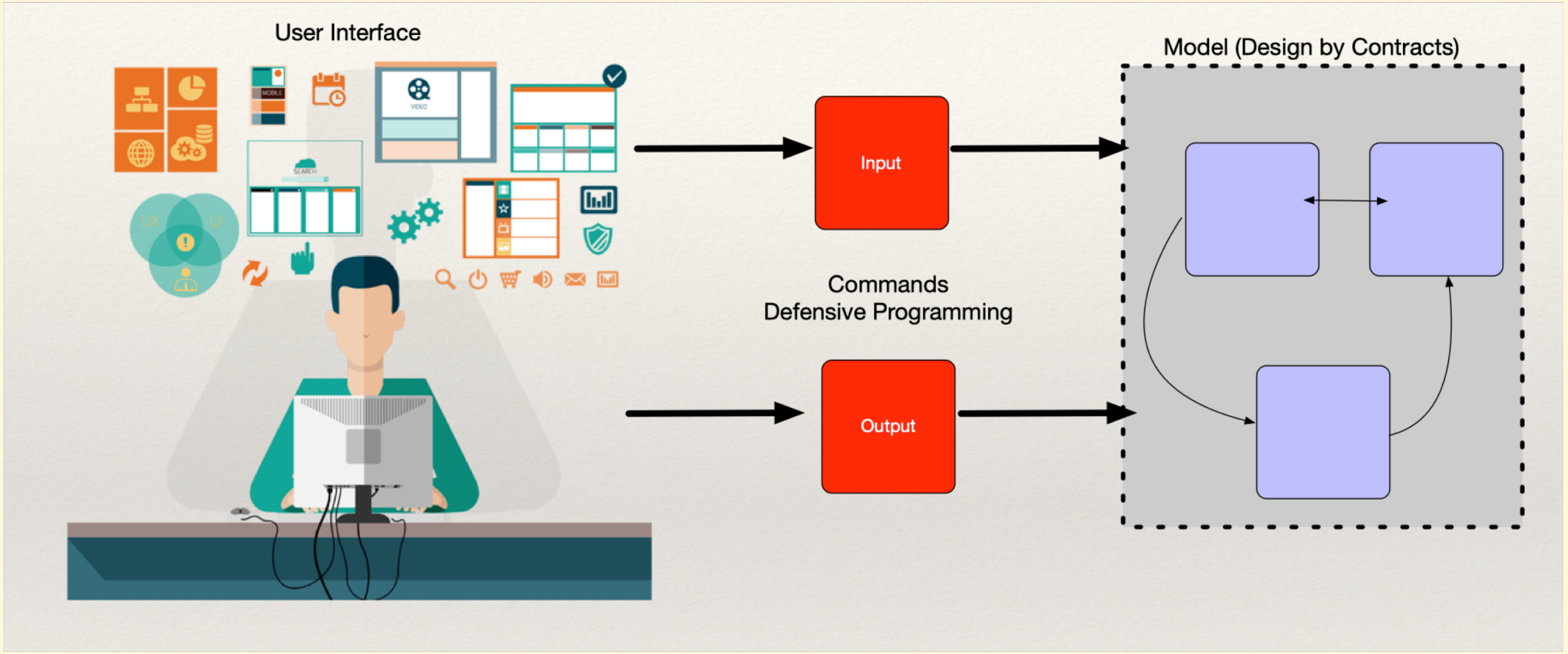
# What should ETF_DEPOSIT do with a semantic violation?

Demanding Contract in Model

```
deposit(a_id: STRING ; a_value: VALUE)
    require
        -- a_id ∈ account.domain
        a_value > "0.0"
    do
        id := a_id
        balance := balance + a_value
    end
```

Tolerant Contract at the User Interface

```
class ETF_DEPOSIT feature
   deposit(a_id: STRING ; a_value: VALUE)
      do model.deposit (a_id, a_value) … end
```


User Interface
Model (Design by Contracts)
Input
Commands
Defensive Programming
Output

# Separation of Concerns

- The User Interface
  - It may be premature to design the UI.
  - May be a web, mobile app, or desktop app.
- The Business Logic (Model)
  - The structure of how the bank organizes knowledge of accounts and transactions
  - The Model should not be dependent on the View, Input or Output

# User Commands Abstractly

- *new*(id: STRING; name: STRING)
- *deposit*(id: STRING; amount: VALUE)
- *withdraw*(id: STRING; amount: VALUE)
- *transfer*(id1: STRING; id2: STRING; amount: VALUE)

Could be a Model Value rather than a STRING

- Notice that we may invoke a sequence of the above from the command line to test the system
- Requirements must be Testable
- Later we can design a concrete mobile, web, desktop UI

# Acceptance Test?

Can write this test before Design & Coding

```
system small_bank
-- allows Tellers to interact with bank

-- Teller inputs
new(id: STRING; name: STRING)

deposit(id: STRING; amount: VALUE)

withdraw(id: STRING; amount: VALUE)

transfer(id1: STRING; id2: STRING; amount: VALUE)
```

```
-- Acceptance Test
-- Create an account and deposit money in it
-- then withdraw some of it

new("B002-75b", "Buffet, Warren")
deposit("B002-75b", 75.20)
withdraw("B002-75b", 5.20)
```

# Acceptance Test Input vs. Output?

Can write this test before Design & Coding

But what should the Output be?

```
-- Acceptance Test
-- Create an account and deposit money in it
-- then withdraw some of it

new("B002-75b", "Buffet, Warren")
deposit("B002-75b", 75.20)
withdraw("B002-75b", 5.20)
```

# What is the output?

# Design Principles

- A class should be known by its interface, which specifies the services offered independently of their implementation
- Class designers should strive for simple, coherent interfaces
- A key issue in module design: which features should be exported, and which should remain secret
- exported feature to a class must be relevant to the data abstraction represented by the class and compatible with other features
- Exported features must maintain the **invariant**
- **Documentation principle:** write software so that it includes all the elements needed for its documentation, recognizable by the tools that are available to extract documentation elements automatically at various levels of abstraction (e.g. contract view vs. text view vs. BON/UML class diagrams)
- Do regular rigorous regression testing (unit/acceptance tests)
- Good style (OOSC chapter 26)

# EHealth Case study: See paper in SVN

| | |
|---|---|
| ENV1 | Physicians prescribes medications to *patients* |

| | |
|---|---|
| ENV2 | There exists pairs of medications that when taken together have dangerous *interactions* |

For example, warfarin and aspirin both increase anti-coagulation.

| | |
|---|---|
| ENV3 | If one *medication* interacts with another, then the reverse also applies (Symmetry) |

| | |
|---|---|
| ENV4 | A medication does not interact with itself (Irreflexive) |

| | |
|---|---|
| REQ5 | The system shall maintain records of dangerous medication interactions |

| | |
|---|---|
| REQ6 | The system shall maintain records of patient *prescriptions*. No prescription may have a dangerous interaction |

| | |
|---|---|
| REQ7 | Physicians shall be allowed to add a medication to a patient's prescription, provided it does not result in a dangerous interaction. |

| | |
|---|---|
| REQ8 | It shall be possible to add a new medication interaction to the records, provided that it does not result in a dangerous interaction. |

# ETF Abstract UI Grammar

```
system ehealth
-- manage prescriptions for physicians and patients

type ID_MD = INT -- physicians
type ID_PT = INT -- patients
type ID_RX = INT -- prescriptions
type ID_MN = INT -- medications

type NAME = STRING
  -- names of physicians, patients and medications

type KIND = {pill, liquid}
  -- for a pill, it is a positive real in mg.
  -- for a liquid it is a positive real in cc.

type MEDICATION =
  TUPLE [name: NAME; kind: KIND; low: VALUE; hi: VALUE]

type PHYSICIAN = {generalist, specialist}

-- User Actions
add_physician (id: ID_MD; name: NAME; kind: PHYSICIAN)
add_patient (id: ID_PT; name: NAME)
add_medication (id: ID_MN; medicine: MEDICATION)
add_interaction (id1:ID_MN;id2:ID_MN)

new_prescription (id: ID_RX; doctor: ID_MD; patient: ID_PT)
add_medicine (id: ID_RX; medicine:ID_MN; dose: VALUE)
remove_medicine (id: ID_RX; medicine:ID_MN)
```

Primitive Types
INT, REAL, VALUE,
CHAR, STRING

Set Enumeration

TUPLE
(record)

Can also have an
array of tuples
(in TLA+: sequence of
records)

User Inputs
(events, commands, actions)

# Simpler example

**system** *ehealth*
—— manage prescriptions for physicians and patients

**type** *MEDICATION* = *STRING*
**type** *PATIENT* = *STRING*

*add_patient* (*p*: *PATIENT*)
*add_medication* (*m*: *MEDICATION*)
*add_interaction* (*m1*: *MEDICATION*; *m2*: *MEDICATION*)
*add_prescription* (*p*: *PATIENT*; *m*: *MEDICATION*)
*remove_interaction* (*m1*: *MEDICATION*; *m2*: *MEDICATION*)
*remove_prescription* (*p*: *PATIENT*; *m*:*MEDICATION*))

# Acceptance Test (based on a Use Case)

```
->add_prescription("p1","m1")
 state 13
 patients:       {p1,p2,p3}
 medications:    {m1,m2,m3,m4}
 interactions:   {m1->m2,m2->m1,m2->m4,m4->m2}
 prescriptions: {p1->m1}
...
->add_prescription("p3","m4")
 state 17 Error e4: this prescription dangerous
 patients:       {p1,p2,p3}
 medications:    {m1,m2,m3,m4}
 interactions:   {m1->m2,m2->m1,m2->m4,m4->m2}
 prescriptions: {p1->m1,m3; p3->m2}
->remove_interaction("m2","m4")
 state 18
 patients:       {p1,p2,p3}
 medications:    {m1,m2,m3,m4}
 interactions:   {m1->m2,m2->m1}
 prescriptions: {p1->m1,m3; p3->m2}
->add_prescription("p3","m4")
 state 19
 patients:       {p1,p2,p3}
 medications:    {m1,m2,m3,m4}
 interactions:   {m1->m2,m2->m1}
 prescriptions: {p1->m1,m3; p3->m2,m4}
```

ASCII representation of Abstract State

Meaningful Error Message

- Program shall not Crash!
- Preserve System Safety Invariant

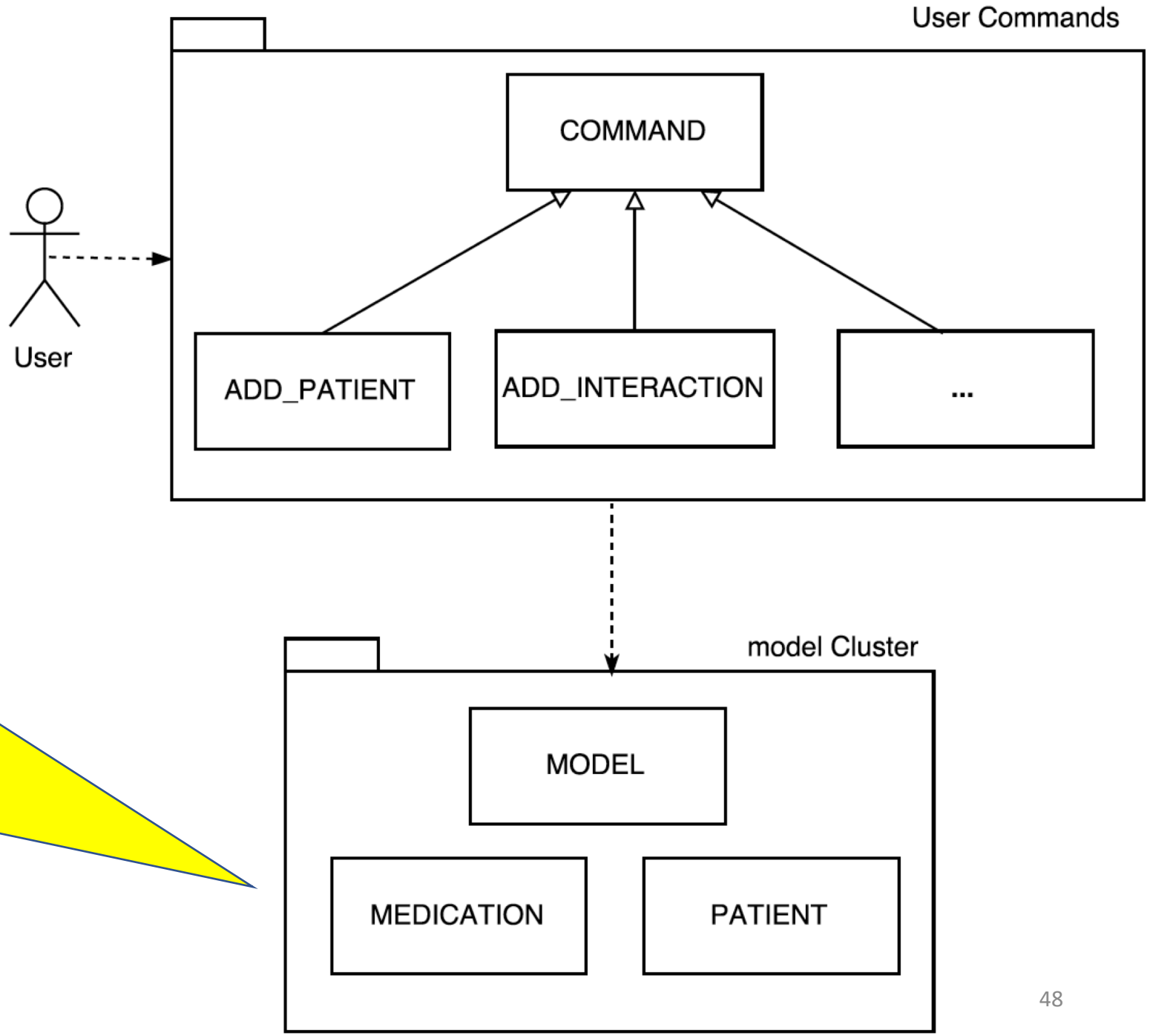
ASCII Abstract State

- Can test via input/output without knowledge of implementation details
- Customer can read it (not Java or C++)

# ETF generated code

Business Logic is Decoupled from User Inputs

- Acceptance Test can be written before Design and Implementation



# MathModels

```
class SET [G] create
  make_empty
feature -- commands with efficient implementation
  choose_item -- choose an arbitrary element
    require not is_empty
    ensure has (item) and chosen
  remove_item
    require not is_empty and chosen
  extend (g: G)
    -- Extend the current set by 'g'.
  union (other: SET[G])
    -- Union of the current set and 'other'.
  subtract (g: G)
   -- Subtract the current set by 'g'.
  difference (other: SET[G])
    -- Difference of the current set and 'other'.
feature -- immutable queries
  chosen: BOOLEAN
  item: G -- an arbitrary member
    require not is_empty and chosen
  count alias "#": INTEGER
    -- Return the cardinality of the set.
  is_empty: BOOLEAN
    -- Is the set empty?
  has (g: G): BOOLEAN
    -- Does the set contain 'g'?
  extended alias "+" (g: G): SET[G]
    -- Return a new set representing the addition of 'g' to Current
  unioned alias "|\/|" (other: SET[G]): SET[G]
    -- Return a new set representing the union of Current and 'other'
  subtracted alias "-" (g: G): SET[G]
    -- Return a new set representing the subtraction of 'g' from Current
  differenced alias "|\" (other: SET[G]): SET[G]
    -- Return a new set representing the difference between Current and
       'other'.
  ...
end
```

*union* (*other*: *SET*[*G*])
-- Union of the current set and 'other'.

Commands

*unioned* **alias** "|\/|" (*other*: *SET*[*G*]): *SET*[*G*]
-- Return a new set representing the union of Current and 'other'

Immutable Queries

```
class interface
  STACK [G]
create
  make

feature -- model
  model: SEQ [G] -- abstraction function

feature -- queries
  count: INTEGER -- number of items in stack
    ensure Result = model.count

  top: G -- top of stack
    require count > 0
    ensure Result ~ model [1]

feature -- commands
  pop -- pop top of stack
    require count > 0
    ensure model ~ old model.tail

  push (x: G) -- push 'x' on to the stack
    ensure model ~ (x ◁ old model)

invariant
      model.count = implementation.count
      -- implementation not shown
end
```

# The abstraction function "glues" the implementation to model

ABSTRACTION FUNCTION: *model*

```
implementation: ARRAY [G]

model: SEQ [G] -- abstraction function
  do
    create Result.make_empty
    from i := implementation.lower
    until i > implementation.upper
    loop
      Result.prepend (implementation[i])
      i := i + 1
    end
  end
```

# A different abstraction function for a linked list

```
1   class
2       HEALTH_SYSTEM
3   feature -- queries ...
4       patients: SET [PATIENT]
5           -- set of patients
6       medications: SET [MEDICATION]
7           -- set of medications
8       prescriptions: REL [PATIENT, MEDICATION]
9           -- prescriptions
10      interactions: SET [INTERACTION]
11          -- dangerous interactions
```

```
50  invariant
51    symmetry_ENV3:
52      across medications as m1 all
53      across medications as m2 all
54        interactions.has (m1.item ↦ m2.item)
55        = interactions.has (m2.item ↦ m1.item)
56      end end
57    irreflexivity_ENV4:
58      across medications as m1 all
59        not interactions.has (m1.item ↦ m1.item)
60      end
61    no_dangerous_interactions_REQ6:
62      across prescriptions.domain as p all
63      across prescriptions[p.item] as m1 all
64      across prescriptions[p.item] as m2 all
65        interactions.has (m1.item ↦ m2.item) and m1.item ≠ m2.item
66        implies
67        not(prescriptions.has([p.item,m1.item])
68        and prescriptions.has([p.item,m2.item]))
69      end
70      end
71      end
72    consistent_domain:
73      prescriptions.domain ⊆ patients
74  end
```

$$Symmetry \triangleq \forall\, m1,\, m2 \in M :$$
$$\langle m1,\, m2\rangle \in interactions \equiv$$
$$\langle m2,\, m1\rangle \in interactions$$

All prescriptions are safe

*i.e.* do not have dangerous interactions

$$SafePrescriptions \triangleq \forall\, m1,\, m2 \in M,\, p \in P :$$
$$\langle m1,\, m2\rangle \in interactions \Rightarrow$$
$$\neg(\langle p,\, m1\rangle \in prescriptions$$
$$\wedge \langle p,\, m2\rangle \in prescriptions)$$

```
31  add_prescription (p: PATIENT; m: MEDICATION)
32        -- Add a prescription of 'm1' to 'p1'.
33    require
34      --p ∈ patients
35      patients.has (p)
36      -- m ∉ prescriptions[p]
37      not prescriptions[p].has (m)
38      -- cannot cause a dangerous interaction
39      -- ∀med ∈ prescriptions[p] : (med, m) ∉ interaction
40      across prescriptions[p] as med all
41        not interactions.has(med.item ↦ m)
42      end
43    do
44      prescriptions.extend ([p, m])
45    ensure
46      prescriptions ~ old prescriptions + [p, m]
47      -- UNCHANGED (patients, medications, interactions)
```

Precondition must be strong enough to preserve the system safety invariant

As an example, the generated Eiffel files `etf_add_patient.e` looks as follows:

```
class ADD_PATIENT inherit COMMAND feature
  add_patient(p: STRING)
    local
      l_p: PATIENT
    do
      -- create a patient from the model cluster
      create l_p.make (p)
      if model.patients.has(l_p) then -- error
        model.set_error ("e1: patient already entered")
      else -- update the model by adding the patient
        model.add_patient (l_p)
      end
      -- execute the command in interactive mode
      -- or store it for batch mode
      command.on_change.notify (Current)
    end
```