Using Eiffel Studio (EStudio) for TDD and DbC: a Bank Example

EECS3311 Software Design
Jackie Wang

Summer 2015
Lassonde School of Engineering, York University

Abstract

This document supplements a series of tutorial videos that demonstrate the practice of Design by Contract (DbC) and Test Driven Development (TDD), using a simple bank account project that is built from scratch. Each section summarizes the (incomplete) list of topics covered in a tutorial video. Each section heading links to the associated video tutorial. You are expected to follow these tutorials, and try to reproduce the project and illustrations yourself.

Contents

1 Create a New Project and Set Clusters

- In the file system, create a directory *3311* as the workspace for this course.
- From a terminal, launch EStudio by typing `estudio15.01 &`
- Create an empty project *bank_proj* under the 3311 workspace.
- From the project setting, we observe that by default EStudio considers the project directory *bank_proj* as the current directory (i.e., `.`), and sets it as the *bank_proj* cluster.

  Note. Directories and clusters are different. Each directory refers to a distinct location in the file system. On the other hand, each cluster points to the location of *some* directory. This means that two or more clusters may refer to the same directory.
- In the file system, create three directories *root*, *bank*, and *tests*.
- In the project setting of EStudio:
  - Remove the default *bank_proj* cluster.
  - Include references to the three newly-created directories as clusters.
  - Now any class that is outside the locations pointed by these three clusters is considered as non-existent. This is why we get a compilation error saying there is no root class. If you inspect your file system, the source file `application.e` is stored outside the three directories *root*, *bank*, and *test*. To fix this error, we move `application.e` into the `root` directory.
  - Recompile, and you succeed with no further errors.
  - Add the *ES_TEST* library.

2 Add the ACCOUNT Class

- Add the ACCOUNT class into the *bank* cluster.
- For documentation, put the developer's name and the list of informal requirements in the class header.
- Declare two attributes *balance* and *credit*.
- Specify two class-level invariants to constrain the possible combinations of *balance* and *credit* throughout the life time of any object instantiated from ACCOUNT:
  - The account credit is always non-negative.
  - The account balance never exceeds its credit.

- Add a feature *make(a_credit: INTEGER)* that can act as a constructor.
  - Add a comment to the *make* feature.
  - In principle, there is no precondition for constructors.
  - Specify the postcondtion of *make*.
  - Show contract view.
  - This is indeed **Design by Contract**: even before the body of implementation is written, we already specify *what* it is supposed to establish, by specifying its postcondition and invariant.

    More valuably, these contracts in Eiffel are Boolean expressions that can be evaluated at runtime, and they can thus be checked against at runtime as the implementation body is executed. In the case of a constructor, nothing will be checked before the ACCOUNT object is initialized, but immediately after the constructor's execution terminates, the postcondition and invariant are checked against the resulting state.
  - Add the body implementation that establishes the postcondition and invariant.
- Declare the list of constructors (e.g., *make*) under the *create* clause.
- Use the *feature* keyword to bookmark sections in the program text, so as to enable us to use the feature browser.

3 Add a Class for Testing ACCOUNT

- Add the TEST_ACCOUNT class that inherits from *ES_TEST* into the *tests* cluster.
- Add a *make* feature and declare that as a possible constructor.
- Add a Boolean query *test_account_creation*.
  - Add a call to the *comment* feature (inherited from *ES_TEST*) that documents the purpose of the test.

    Note. The *comment* feature takes a string argument, which must have two parts that are delimited by a colon (i.e., :). The first part is a short name for the test, and the second part is an informative summary of its purpose.
  - To test the ACCOUNT class, we need to create a new object *acc* (declared as a local variable) by instantiating it. A creation instruction is needed for achieving this: *create* {*ACCOUNT*} *acc.make(10)*.

- As far as a client of the ACCOUNT class is concerned, only features that are declared under the *create* clause (e.g., *make*) can be used as a constructor to initialize an ACCOUNT object (e.g., *create* {*ACCOUNT*} *acc.make(10)*). Calling the same creation instruction twice,

  $$create\{ACCOUNT\}acc.make(10)\ ;\ create\{ACCOUNT\}acc.make(20)$$

  means that the variable *acc* is first assigned to the reference for a new ACCOUNT object with credit 10, and is then reassigned to the reference for another new ACCOUNT object with credit 20.
- Features that are declared under the *create* clause can also be called (e.g., *acc.make(10)*) after the object is created (without re-creating a new object). That is, making the same feature call twice,

  $$\{ACCOUNT\}acc.make(10)\ ;\ \{ACCOUNT\}acc.make(20)$$

  means that the variable *acc* remains the reference, throughout the two calls of *acc.make*, for the same ACCOUNT object, and its credit gets first set to 10 and then set to 20.
- If feature *make* is not declared under *create*, then it can only be called just like a normal feature (e.g., *acc.make(10)*).
- Since *test_account_creation* is a Boolean query, a keyword *Result* is reserved to denote the return value of this query. This special Boolean variable is automatically initialized to *False* and may get re-assigned multiple times before *test_account_creation* terminates: the last assigned value of *Result* upon *test_account_creation*'s termination denotes its return value.
- When *test_account_creation* is used as a test case (by being added as a Boolean test case in the *make* constructor of *TEST_ACCOUNT*), then a test run is considered as a **pass** if it returns *True*; otherwise, if either it returns *False*, or some contract violation occurs before it returns, then it is considered as a **failure**.
- If the *Result* is not explicitly assigned, it remains *False* as its default.

- Up to now, *test_account_creation* has not been chosen as a test case to run.
- Add *test_account_creation* as a test case in the *make* feature of *TEST_ACCOUNT*.
- Similar to the case of *ACCOUNT*, we declare *make* as a possible constructor for *TEST_ACCOUNT*.
- Go to the root class *APPLICATION*, change its parent to *ES_SUITE*, so that we can add all tests that are defined in *TEST_ACCOUNT*.

  We shall anticipate that there will be more test classes to come (e.g., *TEST_TRANSACTION*, *TEST_CUSTOMER*, etc.). Each test class is

responsible a particular unit (i.e., a class). The *APPLICATION* being an *ES_SUITE* allows us to accumulate all tests from all these classes. Each time there is some change made to the bank project, the entire suite of tests must be re-run to make sure that the new change does not introduce a bug. This is called regression testing!

- Write `run_espec` and `show_browser` at the end of the *make* feature of *APPLICATION*.
- Run all tests by running the workbench system.
  - It fails, because the *Result* of *test_account_creation* is never assigned, and will thus remain *False* as its default.
  - Let's try the two extremes, by setting the *Result* of *test_account_creation* as *True* (to pass the test) and as *False* (to fail the test).
- This is indeed **Test Driven Development**: we have only partially developed the system (a single constructor of ACCOUNT), but we have already set up the infrastructure for testing its correctness.

  Note. Of course, so far, the tests are not yet meaningful, but they do illustrate, from the perspective of a client of ACCOUNT, as to how the provided services can be used (by having compatible argument values in feature calls).

4 How a Test Case Fails

- It is convenient to group similar mini-test cases in the same test case (e.g., *test_account_creation*) by reassigning the value of *Result* multiple times.
- Immediately after each re-assignment, we must use an in-code *check* assertion to see if that mini-test case passes. If the *check* assertion fails, then a contract violation occurs and the test case fails without proceeding to further mini-test cases. That is, when there are multiple mini-test cases, the overall test case passes if and only if all of them pass. If *check* assertions were not placed at each re-assignment of *Result*, then the results of all mini-test cases, except for the last one, would be ignored.
- When the Boolean query *test_account_creation* is added as a test case, its returned value (denoted by the keyword *Result*), may get re-assigned multiple times in its body of implementation. At runtime:
  - When executing the body implementation of *test_account_creation*, a contract violation may occur in one of three ways:
    1. An in-code *check* assertion fails. This means that the assumption of the supplier of *test_account_creation* is not satisfied.

2. A violation of the *precondition* of some feature that *test_account_creation* calls (e.g., calling *acc.withdraw(10)* when *acc.credit* is not sufficient). This means that the assumption of the supplier of *withdraw* is not satisfied (note that satisfying this assumption is the obligation of *test_account_creation*, but benefit of *withdraw*).
3. A violation of the *postcondition* of some feature that *test_account_creation* calls (e.g., calling *acc.withdraw(10)* but the resulting balance of *acc* has not been updated properly). This means that the guarantee of the supplier of *withdraw* is not satisfied (note that satisfying this guarantee is the benefit of *test_account_creation*, but obligation of *withdraw*).

– If there is no contract violation until the end of the execution of the query's implementation body, then the test result depends on its Boolean returned value: if it's *True*, then it's a *pass*; if it's *False*, then it's a *failure*.

5 Use Breakpoints and Debugger

- When the test report on the web browser shows a **red bar**, set a *break point* to each of the failing tests. A break point will cause the execution of *test_account_creation*'s body implementation to pause there:
  - We may go **one step at a time** in the context of *test_account_creation*.
  - We may **step into** the context of certain feature that *test_account_creation* calls, then from there we may either go one step at a time or step in further into some other supplier features, and so on.
- Learn how to read the state snapshot at each step.
  - investigate values of variables
  - investigate values of expressions
- The use of break points and debugger should help you fix your code, such that re-running all tests will give you a **green bar**.

6 Specify Contracts for Withdraw

- Add a feature *withdraw(a: INTEGER)* that withdraws some amount a from the current account.
  - Add the precondition

    $$not\_too\_small : a > 0$$

    and postcondition

    $$balance\_set : balance = \textbf{old}\ balance - a$$

  - Show the contract view.
  - Add a new test query *test_withdraw* to *TEST_ACCOUNT*, add it as a new Boolean test case in *TEST_ACCOUNT*, and re-run all tests.
  - Add the implementation:

    ```
    balance := balance - a
    ```

  - The precondition $not\_too\_small : a > 0$ alone is *too weak*
    - This is because it allows inputs values that can cause the resulting (post-) state to violate the postcondition or invariant.

      For example, consider the state where $acc.balance = 0 \wedge acc.credit = 10$ and we call $acc.withdraw(11)$, then the invariant is violated upon its termination.
    - Convert this example into a violation test case using the procedure *test_withdraw_precondition_not_too_weak*.
    - This test case expects a *precondition violation*, so we will use *add_violation_case_with_tag* from ES_TEST.
    - **Fix:** Add another precondition: $not\_too\_big : a < balance + credit$.
  - The new precondition $not\_too\_big : a < balance + credit$ is *too strong*
    - This is because it disallows legitimate input values that should not cause the resulting (post-) state to violate the postcondition or invariant.

      For example, consider the state where $acc.balance = 0 \wedge acc.credit = 10$ and we call $acc.withdraw(10)$, where the expected resulting balance -10 will not violate the invariant *balance_not_exceeding_credit* evaluates to $True$), but we will get a precondition violation because $10 < 0 + 10$ evaluates to $False$.
    - Convert this example into a Boolean test case using the query *test_withdraw_precondition_not_too_strong*.
    - This test case does not expect a *precondition violation*, so we will use *add_boolean_case* from ES_TEST.

Fix: Weaken the precondition: $not\_too\_big : a \leq balance + credit$.

- So we end up with the following preconditions for *withdraw*:

$$not\_too\_small : a > 0$$
$$not\_too\_big : a \leq balance + credit$$

- The postcondition $balance\_set : balance = \textbf{old}\ balance - a$ alone is *too weak*
  - This is because a wrong implementation from the supplier can still satisfy it.
  - For example, consider the state where $acc.balance = 0 \wedge acc.credit = 10$ and the supplier implements *withdraw* by

```
balance := balance - a
if balance < 0 then
  credit := -balance
end
```

    This apparently wrong implementation will always satisfy the postcondition $balance = \textbf{old}\ balance - a$ (since the postcondition constrains nothing about *credit*) and the invariant *balance_not_exceeding_credit*.
  - Convert this example into a Boolean test case using a query *test_withdraw_postcondition_not_too_weak*.
  - We only expect some postcondition violation from a wrong implementation, so we will use *add_boolean_case* from *ES_TEST* (so that when we change back to the correct implementation, this test case will pass).
  - **Fix:** Add a postcondition:

$$credit\_set : credit = \textbf{old}\ credit$$

- So we end up with the following postconditions for *withdraw*:

$$balance\_set : balance = \textbf{old}\ balance - a$$
$$credit\_set : credit = \textbf{old}\ credit$$

7 Add Transaction and Date into Context

- Add the *time* library from project setting
- Add the TRANSACTION class:
  - attributes *value* and *date*
  - invariant
  - constructor *make* (postcondition and implementation)
- Extend the ACCOUNT class:

  - add attributes *deposits* and *withdrawals*
  - change *make* that initializes the *deposits* and *withdrawals* (of type *LIST[TRANSACTION]*)
    * Use pick and drop to see the descendent classes of *LIST*
    * To initialize: *create* {*LINKED_LIST[TRANSACTION]*} *deposits.make*
    * Postcondition: *deposits.is_empty* or *deposits.count = 0*
  - change *withdraw* that updates the *withdrawals*
  - add command *deposit*
  - add command *withdraw_on_date (a: INTEGER; d: DATE)*
    * Iterate through a *LIST*
  - add query *withdraws_on (d: DATE): ARRAY[TRANSACTION]*
    * Initialize an empty *ARRAY* and expand it.
  - add query *withdraws_today: INTEGER* which makes use of *withdrawals_on*
    * Iterate through an array.

- Extend the TEST_ACCOUNT class:
  - add a new Boolean test case: *test_transaction_value_and_date*

- Debug
  - wrong use of the *force* feature of ARRAY
  - no set up for *object_comparison* for the *Result* of *withdrawals_on*
  - no redefinition of *is_equal* in *TRANSACTION*

8 Uniform Access of Account Balance

- Uniform Access Principle
- The *balance* feature, whether implemented by the supplier using computation (as a function) or storage (as an attribute), means the same to the client: the net value from the past deposits and withdrawals.
- As far as the client of ACCOUNT is concerned:
  - The term *acc.balance* represents the net value of the account *acc* as a consequence of its history of deposits and withdrawals.
  - As long as all tests regarding the use of *balance* pass, how the net value is calculated is irrelevant.
- As far as the supplier of ACCOUNT is concerned:

- How the feature of *balance* is implemented is a secret that is hidden from the clients, and the mechanism of calculation may change without affecting the clients. This is an example of **information hiding**.
- When *balance* is affected by many other ACCOUNT features, but not accessed frequently, then the supplier shall implement the *balance* feature by computation to avoid the maintenance of storage consistency.
- When *balance* is accessed frequently and the history lists of deposits and withdrawals are substantial, then the supplier shall implement the *balance* feature by storage to save the cost of computation.