Incorporating TDD and BDD

We will start by demonstrate the TDD process through the implementation of a case study. In this section, we will develop a Mat class to perform 2D matrix algebra, just like we do in the Matlab. This is a class template that can hold an m-by-n matrix for all data types. The matrix algebra includes adding, subtracting, multiplying, and dividing matrices, and it also has element operation abilities.

Step 1 – Writing a failing test

To begin, we will only need the following:

1. Create a `Mat` object from a given a number of rows and cols (the default should be 0-by-0, which is an empty matrix).
2. Print its elements row by row.
3. Get the matrix size from `rows()` and `cols()`.

Based on these requirements, we can have failing unit testing code to boost UTF, as in https://github.com/PacktPublishing/Expert-C-2nd-edition/blob/main/Chapter14/ch14_tdd_Boost_UTF1.cpp.

Now that our testing code is ready, we are ready to develop the code.

Step 2 – Developing code to let the test pass

One way to implement a minimal code segment is to pass the preceding test, as follows at https://github.com/PacktPublishing/Expert-C-2nd-edition/blob/main/Chapter14/ch14_tdd_v1.h.

The preceding code is to define a template class `Mat` that represents a matrix. It provides basic functionality such as creating a matrix, accessing the number of rows and columns, printing the matrix, and managing the underlying buffer for storing matrix elements. Additionally, it includes protection measures like the deletion of copy constructors and assignment operators to ensure proper handling of object copying and assignment.

Once we have the preceding header file, we can develop its corresponding `cpp` file, as follows at https://github.com/PacktPublishing/Expert-C-2nd-edition/blob/main/Chapter14/ch14_tdd_v1.cpp.

Let's say we build and execute it using g++, which supports `-std=c++11` or higher:

```
~/wus1/Chapter-13$ g++ -g ch14_tdd_boost_UTF1.cpp
~/wus1/Chapter-13$ a.out
```

This will result in the following output:

```
Running 1 test case...
  int x=2 x 3[
  1060438054, 1, 4348032,
  0, 4582960, 0,
  ]
  float y=0 x 0[
  ]
  char z=1 x 10[
  s,s,s,s,s,s,s,s,s,s,
  ]
```

In `test_case1`, we created three matrices and tested the `rows()`, `cols()`, and `print()` functions. The first one is a 2x3 `int` type matrix. Since it is not initialized, the values of its elements are unpredictable, which is why we can see these random numbers from `print()`. We also passed the `rows()` and `cols()` test at this point (no errors from the two `BOOST_TEST()` `calls`). The second one is an empty float type matrix; its `print()` function gives nothing, and both its `cols()` and `rows()` are zeros. Finally, the third one is a 1x10 `char` type uninitialized matrix. Again, all the outputs of the three functions are as expected.

Step 3 – Refactoring

So far, so good – we passed the test! However, after showing the preceding result to our customer, he/she may ask us to add two more interfaces, such as the following:

- Create an m x n matrix with a given initial value for all elements.
- Add `numel()` to return the total number of elements of the matrix.

- Add `empty()`, which returns true if the matrix either has zero rows or zero columns and false otherwise.

Once we've added the second test case to our test suite, the overall refactorized test code will be as follows at https://github.com/PacktPublishing/Expert-C-2nd-edition/blob/main/Chapter14/ch14_tdd_Boost_UTF2.cpp.

The next step is to modify the code to pass this new test plan. For brevity, we won't print the `ch14_tdd_v2.h` and `ch14_tdd_v2.cpp` files here. You can download them from this book's `GitHub` repository. After building and executing `ch14_tdd_Boost_UTF2.cpp`, we get the following output:

```
Running 2 test cases...
int x=2x3[
1057685542, 1, 1005696,
0, 1240624, 0,
]
int x=2x3[
10, 10, 10,
10, 10, 10,
]
../Chapter-13/ch14_tdd_Boost_UTF2.cpp(34): error: in
"tdd_suite/test_case2": che
ck x.empty() has failed [(bool)0 is false]
```

In the first output, since we just defined a 2x3 integer matrix and did not initialize it in `test_case1`, the undefined behavior – that is, six random numbers – is printed out. The second output comes from `test_case2`, where all six elements of `x` are initialized to 10.

After we've done a show and tell of the preceding result, our customer may ask us to add other new features or modify the currently existing ones. But, after a few iterations, eventually, we will reach the *happy point* and stop factorizing.