UDC: 004.78 (045)

B.G. Maslovskiy, candidate of technical sciences,
O.M. Glazok, candidate of technical sciences,
(National Aviation University, Ukraine)

MODELING THE OPERATION OF AN AIRCRAFT ON THE BASIS OF TEST-DRIVEN DEVELOPMENT

The way of building the model of the aircraft operation is considered. The approach of test-driven development proposes the possibility to garantee the quality of the code and its easy maintenance and upgrading afterwards. The fee for these features has the form of some additional labour that the developer needs to put in his or her work during coding.

A programmatic code is considered as bad code, if it does not work. However, if a program performs, at least, some activity tat seems to be correct, and the programmer believes that his code works well and is good, what is his confidence based on? Developers often run into a number of problems: there can be errors in the programs, a code can be heavy for understanding, et cetera. And majority from these problems can reveal themselves not immediately, but after weeks, months and even years of work with a project.

Further, even if a programmer is sure that there are no errors in a code, it can happen that it will be necessary to do refactoring, that is, to change something in the code, or, maybe, split large functions into smaller ones. How is it possible after bringing of such changes to acquire the confidence that the obtained code still works? For such a confidence it is necessary to have warranties; these warranties may have a form of successful realization of certain tests. Among the stages of the software lifecycle, such as analysis and forming of requirements, development of specification of product, coding, testing of the product, maintaining the prepared product and correction of the errors discovered in it – the stage of testing takes at least 40-45% of the developer's working time.

Therefore modern developers consider even working code as a bad one, if it is not easily subject to testing. The automatic testing is here implied, that is, the use of testing programs, checking out code in the automatic mode. If it is required to modify the code somehow in order that it can be tested, – it means that the code is written in a bad manner. Why? Because, again, after these changes there is no guarantee that the code is able to continue working exactly as it was working before these changes. Thus, here is the closed circle.

Methods of improvement of this situation are all based on various kinds of testing. The most known of them are manual code testing, automatic code testing, module testing, integration testing. Any professional programmer pays to testing more or less attention.

In a simple case, a test may consist in the «manual» start of the tested programmatic system with application of some (test) data inputs. The manual testing of the program is a very labour intensive process; the bigger the program becomes, the more difficult is to test it by hand. The manual testing works well for the student programs which consist of 300-400 code lines. But, if you develop the program that occupies a million lines, to test all its features by hand during reasonable time is impossible. And, even if you succeeded in doing that, any improvement or alteration in the program done thereafter will introduce probability to break something, so after the alteration you need to repeat all labour-intensive testing again. Finally, your customer or client purchases the program and notices that there are some situations where the program does not work as it should, while you was not able to cover these situation in your program of manual testing.

It is why there appeared technologies which allow to test the program automatically. One of such technologies is the module testing. Module testing is an automatic process of verification of program units for correctness of their work. Essentially, the developer needs to write for every class with non-trivial functionality the set of tests which check up work of methods of this class. For the facilitation of development of tests the special frameworks has been created. Various code-driven

testing frameworks have come to be known collectively as xUnit. These frameworks allow testing of different elements (units) of software, such as functions and classes. The main advantage of xUnit frameworks is that they provide an automated solution with no need to write the same tests many times, and no need to remember what should be the result of each test. Such frameworks are based on the design by Kent Beck, originally implemented for Smalltalk as SUnit. Later SUnit was ported to Java, where it turned into JUnit. From there, the framework was also ported to other programming languages and development platforms, e.g., CppUnit (for C++), NUnit (for .NET). They are all referred to as xUnit and are usually free, open source software.

Extensions are available to extend xUnit frameworks with additional specialized functionality. Examples of such extensions include XMLUnit, XmlUnit.Xunit, DbUnit, HtmlUnit and HttpUnit.

There are also some other popukar testing frameworks, such as CxxTest and TestNG.

All test frameworks share the following basic component architecture, with some varied implementation details:

1) *Test case* is the most elementary concept or class. All unit tests are inherited from it.

2) *Test fixture* (also known as a test context) is the set of preconditions or state needed to run a test. The developer should set up a known good state before the tests, and return to the original state after the tests.

3) *Test suite* is a set of tests that all share the same fixture. The order of the tests shouldn't matter.

4) *Assertions* are functions or macroses that verify the behavior or the state of the unit under test. Failure of an assertion typically throws an exception, aborting the execution of the current test.

The execution of an individual unit test is described in the code as follows:

setup(); /* First, we should prepare the program's environment to make an isolated sandbox for testing. */

...

/* Body of the test; */

...

teardown(); /* Finally, whether the test succeeds or fails, we should clean up the program's environment to not disturb the operation of the other tests or code. */

The setup() and teardown() methods serve to initialize and clean up test fixtures.

Together with the module testing, the technique of development through testing, or test-driven development (TDD), is widely used. It is a technique of programming, at which at first tests are written for the module, and then the programmer adds to the module functionality which corresponds to these tests. Such an approach allows to substantially promote the quality of the code. The tests manage development, and partially appear as formal requirements to the module. At first we create a test which describes the desired behaviour of the class, and then implement the functionality of class, which satisfies these requirements. These tests can be developed in obedience to a specification, or, occasionally, just the test can play the role of specification.

Another practice of programming that lies close to test-driven development is referred to as test-driving. The test-drive development consists of short cycles, not longer that 10-20 minutes.

Development in the test-driven style includes the followings stages [1].

At the first step a developer extracts a program code from repository. This step guarantees that a developer always deals with the latest version of programmatic code.

The second step is adding of a new test to the set of tests which are already present. If some framework is utillized for writing a test, the test is a function or a method which checks whether the system will realize a new conduct, or whether it contains some error about which the testing command reported. The purpose of test is either to reproduce an error, or specify by means of the programming language some new requirements to the programmatic system. It is very important to write the test before making the corresponding alteration in the code; thus, the test places formally new requirements to the code.

Since a new test is written, the third step is executed – the program is compiled and the tests are run. All of the tests must be executed successfully, except for the latest one, which has been just written, because it imposes a new requirement to the system which it does not satisfy yet.

This step is needed in order to make sure, that our test is successfully integrated in the general system of testing, and indeed allows to reproduce an error. That is, a new test reflects new system requirements correctly.

On the fourth step we must correct a mistake or add new functionality with a minimum of efforts. It is important to add the simplest possible solution which will demand from us to write a minimum of code. The less code is written, the less chances, that we will break something with it. Well, it is difficult enough to make an error at writing 3-4 lines of code; at least, it is more difficult, than to make an error while you write 50-60 lines of code.

On the next step it is necessary to do refactoring – a process of change of underlying structure of the program without the change of its conduct. Naturally, refactoring means that we introduce some changes to the code, together with a new probability to break something in the construction of the program. Therefore further it is again necessary to start a test and make sure that our changes pass them, so we have not broken anything. If something goes wrong at this stage, the problem should be fixed, until the tests are run successfully.

And the last step is uploading of our changes into the repository.

Let us consider this approach on the example of development of class which describes the state of an airplane. It is necessary to realize the Airplane class, possessing the following properties: amount of fuel, amount of passengers onboard, presence of pilots and stewardesses onboard, current state (on earth, sets to flight, flight, landing), passengers seats capacity

A class must support the followings methods: fueling, entering and exit of passengers, entering and exit of members of crew, being on the flight, landing. Prior to writing any actual airplane code, we need to create a set of tests for the class of airplane. Let us create a new test with the use of the CxxTest library, that checks up the correctness of creation of an instance of the Airplane() in terms of correct operation of the class constructor. The code of the test case will be as simple as the following:

```
class AirplaneTestSuite : public CxxTest::TestSuite
{
public:
  void TestAirplaneConstruction()
  {
        Airplane airplane;
  }
};
```

Here we call the constructor of the Airplane class. Obviously, this code will not work, because we did not yet write the Airplane class; but, nevertheless, we must try to compile our program and make sure that it is not compiled. This result tells us that our test is correctly plugged into the program, and its compiling is executed, though it is finishes in an abortive manner. In order to comple the test successfully, we need to add the Airplane class to the program. Now, after the necessary changes in the code and inclusion of the header file, our tests get an access to the source code of the Airplane class. Now the test passes successively.

One of the following CxxTest test suits produced for the purproses of testing of such a class may have the following form:

```
class AirplaneTestSuite : public CxxTest::TestSuite
{
public:
  void TestAirplaneConstruction()
  {
        // 100 – number of seats, 500.0 – fuel tank capacity
```

```
        Airplane airplane(100, 500.0);

        TS_ASSERT_EQUALS(airplane.GetFuel(), 0);
        TS_ASSERT_EQUALS(airplane.GetNumberOfPassengers(), 0);
        TS_ASSERT_EQUALS(airplane.GetNumberOfSeats(), 100);
        TS_ASSERT_EQUALS(airplane.GetNumberOfPilots(), 0);
        TS_ASSERT_EQUALS(airplane.GetNumberOfStewardess(), 0);
        TS_ASSERT_EQUALS(airplane.GetFuelTankCapacity(), 500.0);

        // 50 – number of seats, 400.3 – fuel tank capacity
        Airplane airplane1(50, 400.3);
        TS_ASSERT_EQUALS(airplane1.GetNumberOfSeats(), 50);
        TS_ASSERT_EQUALS(airplane.GetFuelTankCapacity(), 400.3);
    }
};
```

It is easy to notice that this test suite includes a bunch of tests that check the necessary values of the instance's properties. Now, among the other features, we took into account the possibility of fueling of the airplane. Again, we must start with creation tests that check up the state of fueling of airplane, taking into account that the airplane must have another additional property – the capacity of the fuel tank. The airplane can take in some amount of fuel, but not more than the fuel tank allows. So, the amount of fuel will be passed as the parameter of a constructor. Now we need to have for testing at least two instances of airplanes with the fuel tanks of different capacity. At running of the tests, it will immediately turn out that the returned amount of fuel is not equal to the expected value, so the initialisation of the property value is performed in a wrong way, possibly due to some garbage information in the memory. Now this error is easily detected and corrected. Now, when we believe that the code is correct, we run the tests again, and there is again an error, now it is in the assertion TS_ASSERT_EQUALS(airplane.GetFuelTankCapacity(), 400.3). The method of GetFuel() should return the 400,3 litre for the second airplane, while in fact it returned 500 for some reason. It is the very realistic situation, though. So, our test suite allows us to find the errors one after another, until the Airplane has the desired behaviour.

Conclusions

The report describes an example of building the model of the on-flight filling the aircraft tanks with fuel on the basis of the test driven development paradigm. The power of the test driven development lies in the fact that the organization of the process guarantees that the code remains correct in the logical sense at almost any point of the development process. As an addition to the correct code logic, the developer acquires a bunch of automatic tests that may help him to maintain the correctness level at any further work on the code. Also, the manner of solving the problem completely changes, as the developer actually considers and produces a solution at developing the tests, prior to the developing of the tested code.

Nevertheless, any conveniences should be payed for. In this case, the payment is the need to write the tests – and it is just the developer who must wear this burden. Behind that, there is another, more valuable sacrifice – the developer must change the approaches to the problem consideration in his or her mind.

And, finally, as Dijkstra said, "Testing may be utillized for demonstration of presence of errors in programs, but it is not able to prove their absence".

References

1. *Stephens M.* Design driven testing: test smarter, not harder /Matt Stephens, Doug Rosenberg. – Apress, 2010. – 368 p.