Experience of creating a library for testing C# and C++ console applications

Vladimir Krotkov
Faculty of lnformation Technology
Samara National Research University
Samara, Russia
galaktikaonline@yandex.ru

Alexandra Danilenko
Software department
Samara National Research University
Samara, Russia
danilenko.al@gmail.com

***Abstract*—The present work provides a description of functionality of library aimed at simplifying creation of console applications in C# and C++ languages by granting opportunities to use built-in adjusted menu of any level of enclosure, special means for control over variables of the program and tracking of a condition of program entities. The library contains functions for manual and automatic testing and the multilevel analysis of console application's performance. The library also includes auxiliary functionality of random or sample generation of user-defined or standard-typed test data and gives opportunities for exact identification of mistakes made by programmer during development. The library is powered with modern technologies of object-oriented programming and developed according to the advanced architectural and algorithmic concepts.**

Keywords—library of tools, manual and automatic testing, data generation, console applications, C#, C++, parser

I. INTRODUCTION

Modern goals of object-oriented programming include universalization, algorithmization, increasing the abstraction level of program and simplification of operating with complex systems. Separate tools, add-ins and subroutines (code debugging tools, Clang Power Tools, JUnit) are used to test such systems, as well as to check their architecture for correctness. However, it is not always possible to perform a quick and easy check whether the algorithm works correctly, nor to test the reaction of application or software component to user actions [1]. The library described in present work eliminates the need for the programmer to manually test each functional element, allowing him to concentrate directly on writing further interaction between software entities and implementing algorithms [2].

II. MATERIALS AND METHODS

The C # language library offered by the authors differs slightly from the C language library. This is due to some of the functionality available in these languages, the specifics of the implementation of some programming patterns, and the challenges commonly faced by developers in the respective languages.

General features provided by the library in both languages include:

- using the built-in menu;
- automatically generating program data;
- carrying out controlled tests of the particular block of the code;
- obtaining information about runtime errors;
- other opportunities.

A. General library capability

A.1. Built-in menu

The menu is the key feature of the library. When launched, it allows the user to select one of its items represented by lambda functions for execution. The menu has protection from the incorrect selection of an item (relevant for manual testing): if the user choses an incorrect item, they will be asked to make a new choice. It is possible to run a sequence of execution of several items (relevant for automatic and semi-automatic testing). The menu works in three main ways:

- manual - the programmer himself chooses what menu item to start and what actions to make inside it;
- semi-automatic - the programmer enters the sequence of commands which need to be executed, and then the menu executes these commands;
- automatic - the menu itself generates the sequence of commands and necessary data for their execution and then starts the necessary points (currently under development).

In the present library, the menu is represented by class Menu and the menu items - by base class MenuItem. A menu object contains a list of MenuItem and itself inherits from MenuItem. In other words, executable menu items can either be simple items (represented by class CommonMenuItem) or other menus. The menu item list has a tree structure where Menu objects can be treated as branches and CommonMenuItem objects - as leaves. Thus, the Composite design pattern was applied, and was achieved a goal of making the behavior of menu items universal, regardless of the particular class that inherits from MenuItem. This architecture opens up an opportunity to extend the inheritance hierarchy around the MenuItem without changing the code interacting with menu items. Such architecture simplifies the interaction with the tree of MenuItem objects: the standardized interaction interface with menus or menu items is represented by virtual (abstract) method execute(), common to every tree object.

Since the creation of Menu object may be complicated, the architectural solution was to implement the Builder design pattern and write the appropriate class. The purpose of this class is to simplify the creation of a complex Menu object by splitting this creation into a chain of simple build calls.

To make the interaction with menu and its components easier for the developer, and to be able to further add and encapsulate new interactions, Facade design pattern was implemented. It controls access to the menu and its components by providing a user with a convenient interface to interact with the entire component bundle [3].

A.2. Random data generation

The library includes pseudo-random data-generating functions. They can generate both: standard data types (int, double, string) and custom data types. Random numbers are obtained by selecting a random value from the boundaries, which can also be generated randomly or entered manually. Random strings are obtained by generating random positive integers and representing them as characters of a certain encoding.

Generation of user-type random data is currently being under development. This process depends on the library's language because C# and C++ provide different tools for working with template (generic) classes and methods.

A.3. Monitored Code Tests

At the current stage of development, the concept of controlled tests plays a major role. It declares that the programmer should be protected from errors while entering test data so that he can quickly, reliably and conveniently test necessary blocks of code. Therefore, menu and other library items are supervised by the data entry control system. In most cases of incorrect input, the program will continue to work, and the user will receive a warning about incorrect input and/or a re-enter request. The programmer can use built-in checks or determine which input is considered correct. For this purpose, the library has the functionality of entering a line according to a predefined template, so that this line may be used to create a specified object. The developer can use pre-built templates or write his own ones.

However, such functionality might not always be enough. In order to check the data for correctness as fully as possible, a system of matching with additional conditions currently lays under development. The programmer will be offered to use the corresponding function of the library and send to it a checking lambda expression or signature of the additional check function.

In order to complete the concept of controlled tests and fully transfer the retrieval of correct data to automatics, a template (generic) class Parser has been written. Its purpose is to directly retrieve objects of the required types from entered string. The specifics of this class also depend heavily on the language of the library. Parser is used in the library itself to convert strings to numbers. One of the advantages of this approach is a more flexible adjustment of line conversion into objects and the possibility of further configuration of converter classes.

A.4. Obtaining test results

The programmer can configure the menu to run a series of tests for some piece of code. Test results can be obtained as information in the console. If a menu is configured to execute a command sequence, it receives each command from internal command buffer and prints it to the console before execution, thus creating a visibility that the command was entered by user itself. So, a chain of all executed items and results of their work will have been built by the time when the buffer becomes empty. Currently new ways of obtaining test results are being developed. Among the rest functionality, these tests will be able to include information on the number of successful (and not so successful) tests that a block of code has been put through.

A.5. Getting runtime error information

When an error occurs within a menu block, information about the location, time, specifics, and description of the error will be available to the user. The user will also be asked whether to stop the program or continue running it.

A.6. Other opportunities

The library offered by the authors also provides ancillary capabilities dependent on the language for which it is made. For example, in C# version of the library conversion of some containers to a string is one of such capabilities, and for C++ version it is a template function getter of a strictly typed object from the string and a function that allows you to add counter of custom objects in some vector while printing. For more information on the possibilities and differences of this library in C# and C++, see paragraphs B and C.

B. Features of C# version of the present library

The library in C# is more object-oriented and its classes better conform to OOP principles such as SOLID, KISS, DRY, YAGNI. This is the result of the fact that in modern C# it is common to write programs using simpler and easier to adapt concepts than concepts of C++ language by authors opinion. Real tasks set for C# developer include creation of an easy-to-manage open to modifications universal system using simple methods. The present C# library will help the developer solve these problems.

B.1. Features of the menu

The C# menu is more customizable. This was achieved by the universal implementation of the builder: it differs from other builders (and from the C++ builder) in such way that it allows to simultaneously create Menu objects directly in place of items of the current menu. This possibility has been opened by transferring the current constructing menu into a stack and placing a new menu under construction instead of the old one. The top of the stack will be popped in place of the current constructing menu afterwards [4]. As the parent class for any menu item StorageDependentMenuItem has an internal list of arguments represented by the LocalStorage object. This object stores the arguments required to run this item, all of which are wrapped in generic class FlexibleArgument<T>. This wrapper allows you to access wrapped argument by both: reference and value, depending on user's choice. Because of some C# specifics, only functions with signature void func(params Object [] objects) are supported. Referred objects within the function can be manually converted to desired types via default converters, which is not very convenient. It is a disadvantage of the menu in this language. However, for the moment, this is the only possible implementation of such functionality in C# known to the authors.

The menu in C# also passes arguments from the outer layer of the menu to the inner layer in its own way. Each StorageDependentMenuItem object has an array of two tuples, hereinafter referred to as the ArgumentScroll. It contains information on the indexes of objects from the external menu item's LocalStorage which should be passed to internal LocalStorage as well as information on how they should be passed: by reference or by value. When passing from the outer layer (main menu) to the lower layer (one of the menu items), chosen wrapped arguments move along the chain to the lower layer, and their change, at user's choice, will affect only the local copy of these arguments or the entire chain until the first LocalStorage where these arguments have appeared by reference.

This approach allows developer to change the information in ArgumentScroll and in LocalStorage during the execution of the program and to monitor the behavior of the program. In this way, the desired flexibility in menu usage is achieved.

B.2. Specifics of random data generation

In C#, it is not possible to correctly implement an interface with a template non-static method GetRandomData, because the implementation of such interface goes against some principles of modern object-oriented programming [5]. At the moment the search for solutions that would allow correct and convenient generation of user-type random data in C# is being held.

B.3. Specifics of obtaining test results

The menu is able not only to send a sequence of items to a chain execution, but also to act correspondingly when it receives a request from functions contained within menu items that are not related to the library. In particular these requests are console data input requests. The user can manually write (or generate) a chain of commands including commands about entering data into the console and transfer this chain to the menu for execution. This will create the appearance of a completely live interaction with the console. Currently, the functional of coloring console output for better visualization is being under development.

C. Features of the C++ language library

Library in C++ is more flexible and better geared towards non-trivial tasks. It will be useful to the developer who solves optimization problems [6]. The C++ language library can be assembled for use on any UNIX-like system (including Linux), Microsoft Windows, macOS, and on some other systems. This was achieved by using a cross-platform project build system CMake.

C.1. Features of the menu

The menu in C++ has richer inheritance hierarchy: the menu is further divided into a single-run menu Menu and a multirun menu MultiLaunchMenu. MultiLaunchMenu allows the programmer to mark some MenuItem as an exit item and terminate the menu after this item is ran. The menu itself has no flaws of C# menu - CommonMenuItem items are designed to work with lambda functions of the standard library, as are C# menu items, but the standard C++ functional of flexible external context capture in lambdas has made it possible to exclude the storage of argument from menu item, because the user can specify the desired context and call the function they need inside the lambda with captured context [7]. Due to lambda's versatility, it is also possible to create a non-trivial lambda and tweak the behavior of the program.

For user convenience, the present C++ library provides a basic set of functions for building certain types of MenuItem, such as a confirm selection menu. These functions return a builder with a half-built object so that the user does not have to write the same build chain manually.

Builders of the entire inheritance hierarchy do not have a base class, which is a disadvantage. Polymorphic behavior while building a MenuItem object is impossible to organize. Solutions to this problem are currently being sought.

C.2. Code Testing Features

The library namespace contains a globalDataInputMode variable that is responsible for the current global input mode for the entire program. In the future, it will allow to adjust the behavior of library's functions at different settings more qualitatively. At the moment, it can be used to tweak the behavior of the program directly within the tested functions. This variable allows a unique transfer of the library 's testing functionality into the function. This can be used to link the library to the insides of the tested code.

III. Results and Discussion

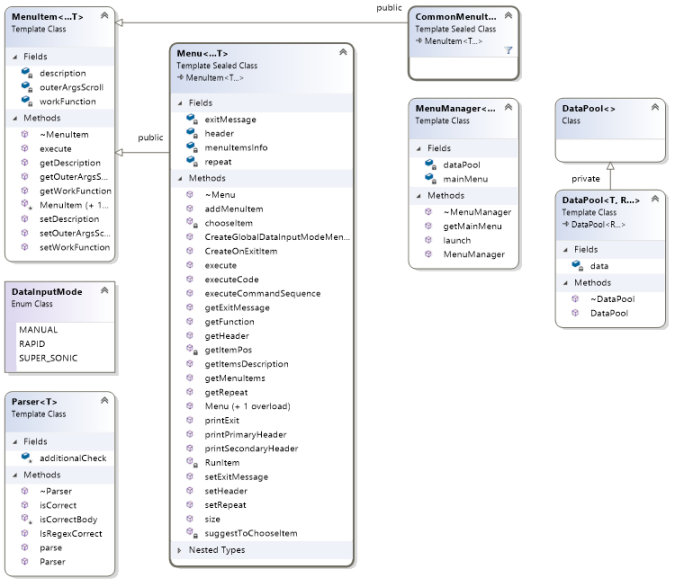

Fig. 1. Diagram of the main relationships between library elements and their composition at particular development stage.

```
int someInteger = 666;
std::string hello = "Hello world!";

auto builder = CommonMenuItemBuilder();

MenuItem *menuItem1 =
    builder
        .buildCode([&someInteger]() { std::cout << someInteger + 1 << '\n'; })
        .buildOptions(Options("1", "print some integer + 1"))
        .build();
MenuItem *menuItem2 =
    CommonMenuItemBuilder()
        .buildCode([&hello]() { std::cout << hello << '\n'; })
        .buildExecuteCommand("2")
        .buildExecuteDescription("print hello")
        .build();

Menu *resultMenu = new MultiLaunchMenu(
    MenuDescription("Main Header", "Goodbye!"), Options("run", "mainMenu"),
    std::vector<MenuItem *>({menuItem1, menuItem2}), 1);

MenuManager manager(resultMenu);
manager.launch();
```

Fig. 2. Process of menu construction on the client side at particular development stage.

```
Библиотека для продвинутого тестирования и отладки
консольных приложений на языке C++
Разработчик: Кротков В.Д., 6213-020302D
Научный руководитель: Даниленко А.Н.
-=-Добро пожаловать в Главное меню-=-
1 -> Первый тестовый пункт - обычный пункт
2 -> Вторый тестовый пункт - обычный пункт
3 -> Третий тестовый пункт - одинарное меню из 2 подпунктов
4 -> Выход
Ваш выбор: 1

Сейчас строка 4234999 пройдет тест на число:
1
С возвращением в Главное меню
1 -> Первый тестовый пункт - обычный пункт
2 -> Вторый тестовый пункт - обычный пункт
3 -> Третий тестовый пункт - одинарное меню из 2 подпунктов
4 -> Выход
Ваш выбор: 4

Выход из меню
```

Fig. 3. Working console menu.

IV. Conclusion

The present library developed by authors gives developer an opportunity to significantly reduce the time needed to write monotonous code, as well as to simplify the task of testing applications. The main advantages of this library are the possibility of performing automated tests of functions and the possibility of obtaining results from these tests. The library was integrated and tested on real tasks.

The functionality of the library is currently being expanded.

References

[1] F. Horváth, B. Vancsics, L. Vidács, Á. Beszédes, D. Tengeri, T. Gergely and T. Gyimóthy, "Test Suite Evaluation using Code Coverage Based Metrics," CEUR Workshop Proceedings, vol. 1525, pp. 46-60, 2015.

[2] B. Oliinyk and V. Oleksiuk, "Automation in software testing, can we automate anything we want?" CEUR Workshop Proceedings, vol. 2546, pp. 224-234, 2019.

[3] E. Freeman, B. Bates, K. Sierra and E. Robson, "Head First Design Patterns: a Brain-Friendly Guide," Saint-Petersburg, 2018, 656 p.

[4] E. Gamma, R. Helm, R. Johnson and J. Vlissides, "Design Patterns Elements of Reusable Object-Oriented Software," Addison-Wesley, 1994, 395 p.

[5] "How to write Regular Expressions?" Geeksforgeeks [Online]. URL: https://www.geeksforgeeks.org/write-regular-expressions/.

[6] A.S. Yumaganov and V.V.Myasnikov, "A method of searching for similar code sequences in executable binary files using a featureless approach," Computer Optics, vol. 41, no. 5, pp. 756-764, 2017. DOI: 10.18287/2412-6179-2017-41-5-756-764.

[7] Lambda expressions [Online]. URL: https://en.cppreference.com/w/cpp/language/lambda.