Adam Zálešák, 493071
Filip Hájek, 493341
Katarína Platková, 493144

PV286 Project - Phase II

Project design

Argument Parsing

The first step of the program is checking the number of arguments and the length of all of the arguments. If they are too many or too long, the exception is thrown. Otherwise, the program continues its execution with provided arguments.

If there is a help option among the switches, help is printed to the console and program returns without doing anything else.

In the other case the arguments with their values are further processed within the parse method. The parse method checks for both the short and long options. All of the possible argument types are listed in the dictionaries. The method goes through all of the options from the program input. It checks whether all of the argument values are not empty, valid and are provided at most once. The ones passing the control are stored in a dictionary where key is argument type and value is list of values represented by string.

If all of the arguments are loaded correctly, the method checks whether the mandatory options (to, from) are provided and whether the values for input and output arguments are valid. It also checks if the arguments for input and output formatting are as expected.

The result of the parse method is the above mentioned dictionary containing options with their values.

Input and Output Streams

After the successful parsing of arguments, the input and output values are retrieved from the resulting dictionary. The stream service checks whether the dictionary contains both the input and output. In case of the file, it checks its existence.

Converter Creation

The next step of the program is the convertor creation. In this step the arguments are taken and passed to the converter factory separately as from, to, delimiter and format specific input and output options.

The factory stores the dictionary of all possible input and output combination formats and the create method provides the rest of the optional arguments. Additionally, the input validator is created and passed to the convertor.

Taking all of these inputs, the factory creates the convertor specified by arguments.

Convert Part

The reading of the input and its conversion is performed in parts. The input comes as Stream to the Convert method implemented in the base abstract class Convertor. Processing and conversion is then handled in parts. The input stream is read by bytes. The first step after reading the byte is checking its validity. This is done using the format specific validator created during the convertor creation. The bytes are stored in a buffer - an array of 4096 bytes. When the buffer is full or the input is finished, it is then passed to the ConvertPart abstract method, which is implemented in each specific converter.

For each conversion, there is a class that inherits from the Convertor class. Its ConvertPart method receives as a parameter part of the input and the output stream (stdout or open file). The ConvertPart methods convert the data, taking into account options (e.g. bit padding).

If the input is found to be invalid during conversion, an exception is thrown with a message about what is wrong with the input data. These exceptions are caught globally in Program.cs, where their message is dumped to the console and the program is terminated.

A special case of converter is when the input and output formats are of the same type. The program then only needs to check whether the provided input is valid. the convert part method then just writes bytes from input to the output stream.

Current progress

Conversions for **bytes**, **hex**, and **bits** formats have been implemented. The application at this stage allows conversion from any of these to any of these, with the ability to use the appropriate "from" and "to" options defined in the assignment. Input can be a file or stdin, output can be a file or stdout.

In the next phases, support for int and array formats will be added. Also, support for custom delimiter will be implemented, which is in progress but not yet fully functional.

Github Actions

On Github Actions we set up two pipelines, integration and deployment. The integration pipeline builds the application and runs tests. The deployment pipeline runs the integration pipeline in the beginning and if it is successful it publishes the app and creates a Nuget package and a new release. The deployment pipeline publishes apps for Linux and also for Windows. The nuget package is uploaded to Github Packages. You can install an app via Nuget using 'dotnet nuget install'. The deployment pipeline also automatically increases the version number.

The integration pipeline runs only on 'feature/*' branches. These are branches for implementing some features. After a branch is reviewed it can be merged to the main branch if the integration pipeline does not fail. When the merge happens, the deployment pipeline runs. We can also push commits into the main branch when the change is small or it is some bug.

In the integration pipeline, we will add steps that are going to run code analysis. Currently, pipelines do not run any code security analysis.

Testing

For testing, we choose the XUnit framework for .NET 7. The tests are divided into unit tests and integration tests. Unit tests are not completed because there was no time to write a unit test for every class and every public method. We changed the codebase multiple times so we would also have to rewrite unit tests after some bigger changes. So we really focus on implementing integration tests that are reusable, easy to use, and mainly not affected by some refactor or major code change.

The integration test runs the whole application with given arguments and data. With XUnit a CliWrap library, it is easy to set up tests like that for C# apps. E.g. when we add a new format, we need to add only one line of code where we specify arguments with input data and expected output data. The new test case is ready to run in a local or Github environment.

Now that we have a fixed code structure, we will add unit tests for the 'Convertor' classes and 'ArgumentParser' class, because they are critical for the application to correctly work. We will also try to get as much coverage as possible. For the 'Convertor' and 'ArgumentParser' classes tests will cover all possible cases and some edge cases.

Problems

TDD

We tried using TDD but we found out that it is not as useful as we thought and that there are more problems than benefits when using this approach, especially in the early stages of the implementation. When writing unit tests we gave up TDD immediately. Every single change of the tested class interface could potentially break the unit test so we would also have to rewrite tests after some code changes. We did not have a solid codespace e.g. we changed the Convertor class multiple times. However, it does make sense to prepare and write tests on the integration test level, because code changes do not affect the test code. But on the other hand, CI/CD pipelines will fail and if someone makes an error it is hard to find the test which failed because many other tests will fail because of the not implemented feature. We are aware that there are some solutions but we agreed that we should write some features and after implementation we test them.

Large inputs

One of the problems encountered was too large input value to convert. Due to that, we decided to read the input stream until the delimiter is encountered. However this would not solve all of the situations with long input. Sometimes, it is possible that there will be a large amount of input bytes until the delimiter.

So the second step to handle those inputs was to create some buffer. The buffer works in a way that if a certain amount of bytes is read, those bytes are processed and written to the output stream. Only after that it is possible to continue with input processing. This should prevent the situations with running out of memory.

Format

Another thing we needed to handle was that the options and input can sometimes be provided in an incorrect format. Specifically, there are a lot of possibilities of assigning incorrect arguments or argument values. Also the input to be converted can contain bytes that are not representing valid characters of given format. Therefore it was necessary to try to handle all of such cases.