FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Assignment of master's thesis

Title:	Refactoring of modular library of compression methods SCT
Student:	Bc. Filip Geletka
Supervisor:	Ing. Radomír Polách
Study program:	Informatics
Branch / specialization:	Web and Software Engineering, specialization Software Engineering
Department:	Department of Software Engineering
Validity:	until the end of winter semester 2022/2023

Instructions

Analyze the current inner workings of the SCT library.
Analyze, redesign and reimplement the modular framework and take into account the current and possible future compression algorithms and their logical chaining in implemented compression methods.
Analyze, design and implement file format for storing compressed data for the current and possible future methods and take into account the modular nature of the library from the previous step and the ability to describe exact parameters of algorithms used during the compression for the purposes of decompression.
Analyze, design and implement testing.

Electronically approved by Ing. Michal Valenta, Ph.D. on 19 February 2021 in Prague.

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Master's thesis

Refactoring of modular library of compression methods SCT

Bc. Filip Geletka

Department of Software Engineering
Supervisor: Ing. Radomír Polách

December 15, 2021

Acknowledgements

I want to thank my thesis supervisor Ing. Radomír Polách. The door to Mr Polách's office was always open whenever I ran into a trouble spot or had a question about my research or writing. I must express my very profound gratitude to my parents and my friends for providing me with unfailing support and continuous encouragement throughout my years of study and through the process of researching and writing this thesis. My accomplishments would not have been possible without them. Thank you.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as a school work under the provisions of Article 60 (1) of the Act.

In Prague on December 15, 2021 . . . . . .

Czech Technical University in Prague
Faculty of Information Technology

Citation of this thesis

Geletka, Filip. *Refactoring of modular library of compression methods SCT.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2021.

Abstrakt

Táto diplomová práca sa zaoberá analýzou, návrhom a realizáciou vylepšení v knižnici kompresných algoritmov SCT. Knižnica SCT - Small Compression Toolkit je modulárna knižnica obsahujúca sadu kompresných algoritmov vyvíjaných v programovacom jazyku Java. Kompresné algoritmy boli do knižnice pridávané postupne ako samostatné moduly, a ich celková integrácia do knižnice SCT je značne obmedzená a nedokončená. Cieľom týchto vylepšení je plná integrácia existujúcich algoritmov do knižnice SCT s dôrazom na ich voľné reťazenie a znovupoužitie. Patričný dôraz je kladený na uľahčenie implementácie budúcich prídavkov do knižnice SCT.

Klíčová slova kompresia, dekompresia, spracovanie dát, modulárna knižnica, refaktoring, konfigurácia

Abstract

This thesis deals with the analysis, design and implementation of improvements in the SCT library of compression algorithms. The SCT - Small Compression Toolkit library is a modular library containing a set of compression algorithms developed in the Java programming language. Compression algorithms were gradually added to the library as separate modules, and their

overall integration into the SCT library is limited and unfinished. The goal of these improvements is the full integration of existing algorithms into the SCT library with an emphasis on their free chaining and reuse. Due emphasis is placed on facilitating the implementation of future additions to the SCT library.

Keywords compression, decompression, data processing, modular library, refactoring, configuration

Contents

List of Figures

Introduction

Motivation

Never before in the history of humanity has so much data been produced and processed in such a short time. Computer data storage getting more affordable and internet connection getting faster and more worldwide available directly influences the volumes of these data. In order to be as efficient and as quick as possible in storing and transferring these data, various compression algorithms have been invented and implemented. However, only a few of them made such an impact in the matter of compression speed, reliability and memory efficiency that they became standardised and widely used in computer science. Through the years, those compression algorithms became a subject of study and further improvements for many individuals. The intention is to study, analyse and understand them and discover new innovative ways of improving them, or just take inspiration when designing completely new techniques of compressing data.

In order to make a study of the fundamental compression algorithms as digestible as possible for new students, a small collection of compression algorithms is being developed and maintained. This collection, called a programming library, aims to aid the teaching of the basic principles of data compression for students engaged in this field of computer science. Interested students can examine the exact behaviour of implemented algorithms, experiment with them by changing essential parameters for their function, or extend them to their liking.

However, this library has been subject to various changes and additions, which often operate only in a closed environment and do not support unrestricted interconnections with other modules. This results in a rapidly growing risk for the overall robustness and functionality of the library. This situation presents an opportunity to reevaluate original ideas and intentions for the library and introduce a new approach to the library's data flow, module chaining, module integration, testing and facilitation of future expansions.

Main goals

The main objective of this thesis is to redesign and implement the core functionalities of the Small Compression Toolkit (SCT for short) library and add new features that will make future additions easier to implement and integrate. The SCT library is a collection of compression algorithms developed at the Department of Theoretical Computer Science at the Faculty of Information Technology at Czech Technical University in Prague [1]. The library is developed in the Java programming language.

The first task is to analyze and understand the library's current design and implemented functionality. The excellent sources for this research are bachelor's and master's theses whose assignments were to add new compression methods. Due to a poor emphasis on version control and regression testing, many implemented modules are non-functioning. The part of the analysis is to determine what components are inoperational and need a fix.

The next challenge is to redesign and implement changes to a module chaining and data flow in the library and provide a unified interface to all modules. Even though it is not a part of the assignment of this thesis, it makes sense that a reasonable effort should be made to fix as many as possible of the non-functioning algorithms. After that is done, a new dedicated file format needs to be implemented to store the output from the SCT library.

The last task is to implement a unit, integration, regression and performance testing. The performance measurements on the provided real-life data shall be presented.

Thesis structure overview

This section briefly explains what is covered in particular chapters and what information can be found in specific chapters.

Chapter 1 is devoted to defining basic notions important for further chapters.

Chapter 2 focuses on the analysis of the SCT library, its contents, its current operational status and the definition of the functional and non-functional requirements.

Chapter 3 is dedicated to the implementation of required functionality, with particular emphasis given to meet functional and non-functional requirements mentioned in Chapter 2. The end of the chapter focuses on new tools for unit, integration and performance testing.

Chapter 4 focuses on performance testing. In this chapter, used corpora are introduced, and compression time and compression ratio are measured on them. Lastly, the newly implemented modular framework is compared to the solution that it is deemed to replace.

Chapter 1

Basic notions

This chapter is dedicated to defining and explaining fundamental terms relevant to the topics addressed in the following chapters of this thesis.

1.1 Alphabet

Alphabet is a finite set of distinguishable symbols. In the scope of this thesis, the Extended ASCII table will be used. In this set of symbols, each symbol is represented by a binary value. This extended version of the ASCII table supports the representation of 256 different symbols. This is because the Extended ASCII uses eight bits to represent a character as opposed to seven in the standard ASCII table. All values of a symbol from the Extended ASCII table can fit one byte in computer memory, thus making it a perfect candidate for usage when working with raw data [2].

1.2 Symbol

Symbol is an element from the alphabet. In the scope of this thesis, one symbol is equivalent to one byte, and since the size of one byte is eight bits, values for a symbol ranges from 00000000 to 11111111 [3].

1.3 String

String is a finite sequence of symbols from the alphabet. In the scope of this thesis, string refers to a finite sequence of bytes [3].

1.4 Codeword

Codeword is a sequence of bits. In other words, a codeword is a string over the binary alphabet [4].

1.5 Code

Code is a system of rules to convert information, such as a symbol or string, into another form or representation. Code substitutes string with codeword from the binary alphabet for the purpose of converting the original information [4].

1.6 Triplet

In the context of this thesis, the *triplet* is understood as an n-tuple - an order set of n elements. Multiple algorithms discussed later in this thesis work with different n-tuple, ranging from 1-tuple up to 3-tuple [5].

1.7 Data compression

Data compression or *encoding* is a process of transforming the input string to the output string of a different format, which in most cases has a shorter length. This can be very beneficial because it reduces the resources required to store and transmit the data. Data compression can be lossless or lossy. *Lossless compression* allows the original data to be completely reconstructed from the compressed data, while *lossy compression* reduces the size of compressed data by omitting unnecessary or less important information [6].

Usage of lossless and lossy data compression varies mainly on the type of the concerned data. For the images, video and audio data, lossy data compression is often preferred because it brings a significant decrease in compressed data length, allowing faster data transfers and less use of computer storage. On the other hand, for most of the other data types, such as text, for example, lossless data compression is required because there is none or close to no information available for the omission [7].

Data compression is subject to a space-time complexity trade-off. It involves trade-offs among various factors, such as the degree of compression, time complexity and the computational resources required to compress and decompress the data [3].

1.8 Data decompression

Data decompression or *decoding* is the process of reversing data compression. This action transforms compressed data to its initial uncompressed form. When lossy compression is applied, it is impossible to entirely reconstruct original data because some information was omitted when initial data compression happened [6].

1.9 Compression algorithm

Compression algorithm or *encoding algorithm* is a finite sequence of steps required for data compression [3].

1.10 Decompression algorithm

Decompression algorithm or *decoding algorithm* is a finite sequence of steps required for data decompression [3].

1.11 Compression method

Compression method is a name used to label a specific compression algorithm together with its associated decompression algorithm [8].

1.12 Model

Model is an internal structure used by compression method which contains information about currently processed input data. For the decompression to be successful, it must use the identical model as was used for the compression. Otherwise, it won't be able to output the matching string as was on the input for the compression [8].

1.13 Adaptive compression method

Adaptive compression method adapts the data model used during compression in accordance with the input data. The compressed data must not include the data model. The decoder builds the same model as the encoder. The encoder and decoder begin with a default model, yielding poor initial data compression, but performance improves as they acquire more information about the processed data. The encoder compresses the next data block first and then adapts the model in accordance with this block. This order enables the decoder to produce an identical model. If the encoder modifies the model before it compresses the block, the decoder can not decompress the data because the next block is compressed using an unknown data model [8].

1.14 Semi-adaptive compression method

Semi-adaptive compression method adapts the data model in accordance with the input data. The compressed data must include the data model. The decoder restores the model first and then decompresses the data using this model. Semi-adaptive methods are mostly two-pass compression methods.

The first pass is used to build the data model, and in the second pass, the data are compressed using information from the first pass [8].

1.15 Context compression method

Context compression method uses the information about the context of the input block during compression. The result of compression is dependent on surrounding blocks [8].

1.16 Dictionary compression method

Dictionary compression method uses a dictionary of phrases during the compression. The dictionary is initialized at the beginning of the compression. During the compression and decompression, the new phrases are added to the dictionary. Dictionary compression methods are adaptive compression methods, and the compressed data includes indices of the phrases in the dictionary [8].

1.17 Compression ratio

Compression ratio measures the relative reduction in the size of data representation produced by a data compression algorithm. This compression ratio is a ratio of the length of compressed data to the original size of data (Formula 1.1) [9].

$$Compression\ ratio = \frac{Length\ of\ compressed\ data}{Length\ of\ original\ data} \tag{1.1}$$

For instance, the compression ratio of 0.75 means that the compression algorithm was able to reduce the compressed file size to 75% of its original size. If the result of this equation is greater than 1, it means compression of the input data resulted in negative compression. This outcome can often be observed when using compression algorithms on small files. Many algorithms need to store necessary information, such as a model, in the compressed data during the compression. The size of these supporting structures, combined with the size of the compressed data, can even surpass the size of the initial input data, thus making the usage of the compression algorithm counterproductive and unnecessary.

1.18 Entropy

Entropy in data compression means the randomness of the data that are subjected to the compression. The higher the entropy, the worse the compression

ratio. That means the more random the data are, the less effective the compression is going to be [10].

1.19 Redundancy

Redundancy is the existence of additional and unnecessary information in the data in the sense that if it were dropped, the data would still be essentially complete, or at least could be completed. Redundancy is related to the degree to which it is possible to compress the data. Lossless data compression reduces the number of bits used to encode data by identifying and eliminating statistical redundancy.

The more redundancy there is in data, the more predictability we have, which means less entropy per encoded symbol and hence the higher compressibility. Since the repeated patterns have been eliminated during the compression, compressed data are less predictable, and the unwanted redundancy has been reduced or entirely eliminated [11].

1.20 Corpus

Corpus is a collection of files commonly used for comparing compression methods. The intend of their usage is to benchmark lossless compression methods to enable researchers to evaluate and compare them [12]. Among the primary attributes which are measured are compression ratio, compression time and decompression time.

1.21 Module

In the scope of this thesis, a *module* is understood as an independent unit responsible for a specific task, for example, a compression algorithm, a decompression algorithm, a data input and output handler or a logger.

Chapter 2

Analysis

This chapter focuses on the analysis of the SCT library, its contents, its current operational status and the definition of the functional and non-functional requirements.

The first couple of sections briefly introduce the SCT library and its purpose, examine the library from the point of view of the used technologies, look at the data input and output handling, explain data flow in the SCT library and closely analyse implemented compression methods. Each compression method and utility from the library is briefly described and accompanied by a UML class diagram. These diagrams were generated with the IntelliJ IDEA. IntelliJ IDEA is IDE developed by JetBrains. Integrated development environment (IDE for short) is a handy tool for software development that can provide many valuable utilities to make work more productive, such as integrated VCS, UML class diagram generator or built-in build automation tool like Apache Maven [13].

The next section assesses the operational status of the implemented modules.

The last two sections of this chapter focus on specifying functional and non-functional requirements resulting from the assignment of this thesis. Emphasis will be given on application requirements and explain what is required and expected from the final product.

2.1 SCT library

As mentioned in the introduction of this thesis, the SCT library is a collection of compression algorithms developed at the Department of Theoretical Computer Science at the Faculty of Information Technology at Czech Technical University in Prague [1]. SCT is an initialism of Small Compression Toolkit. Its primary intention is to help students interested in data compression with their learning and provide a playground for their experiments. The library is developed in the Java programming language. Java was chosen as the main

programming language of this library because of its platform independence, easiness to learn and use, and because the implementation of the first algorithm, ACB, was not available in Java at that time [5].

2.2 Associated theses

As of today, two master's and four bachelor's theses contributing to the SCT library have been published. The following list briefly introduces them. All implemented algorithms in these theses will be covered in the following sections. This master's thesis is not included in the mentioned count nor in the following list.

- **Implementation of the ACB compression method improvements in the Java language -** This master's thesis, written by Jiří Bican in 2017, laid the foundation for the SCT library and added the ACB compression method as the first of many to come. Core utilities of the SCT library still used to this day have been designed and added as a part of this thesis [5].
- **Implementation of the DCA compression method in the Java language -** This bachelor's thesis, written by Jakub Novák in 2018, contributed by adding the DCA compression method [14].
- **Implementation of the LZ77, LZ78 and LZW compression methods in the Java language -** This bachelor's thesis, written by Ladislav Zemek in 2018, contributed by adding the LZ77, LZ78 and LZW compression methods [15].
- **Implementation of the LZY, LZMW and LZAP compression methods in the Java language -** This bachelor's thesis, written by Ján Bobot in 2019, contributed by adding the LZY, LZMW and LZAP compression methods [16].
- **Implementation of BWC compression method and its variants in Java programming language -** Filip Geletka wrote this bachelor's thesis in 2019. Additions include BWT, MTF and RLE compression methods and Arithmetic and Huffman adaptive entropy coders [17].
- **Finite State Entropy Coder for SCT library -** This master's thesis was written by Ladislav Zemek in 2021. Its implementation includes LZFSE and ZStandard compression methods and FSE triplet coder [18].

2.3 Used technologies

This section lists used technologies in the SCT library and emphasises the benefits of their use.

- **Git -** Git is software used for tracking changes and new additions to any set of files. It is a type of version control system (or VSC for short) used during the software development cycle, enabling better cooperation with other team members and bringing new robustness levels to the product. The SCT library uses GitLab, which is a web-based Git repository [19]. It is hosted at the Faculty of Information Technology at Czech Technical University in Prague [1].

- **Apache Maven -** Apache Maven is a software project management and comprehension tool that can manage a project's build, reporting, documentation and testing [20].

- **Apache Commons CLI -** This library provides an API for parsing command line options passed to the SCT library. It's also able to print help messages detailing the options available for a command line tool. Commons CLI supports different types of options and is highly customizable [21].

- **Apache Log4j 2 -** This utility is one of the most widely used logging frameworks used in the Java programming language. This tool is inspired by existing logging solutions, such as predecessor Log4j 1 or java.util.logging. Compared to printing debug information to the standard streams, it allows to change logging level and enables fast and efficient management of the logging files [22].

- **JUnit 5 -** JUnit 5 is the 5th major version of the programmer-friendly testing framework for Java and the JVM. The goal is to create an up-to-date foundation for developer-side testing on the JVM. This includes focusing on Java 8 and above, as well as enabling many different styles of testing. JUnit features include assertions for testing expected results, test fixtures for sharing common test data and test runners for running tests [23].

2.4 Data flow in the library

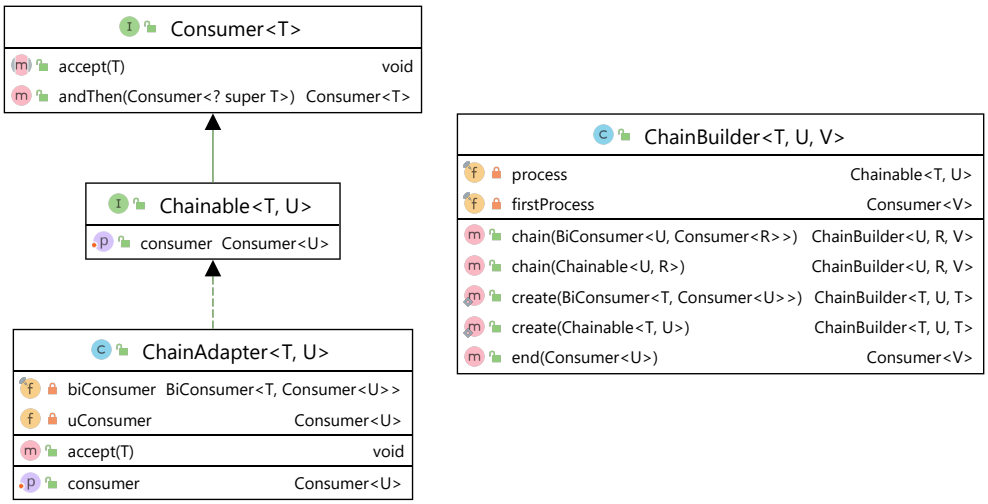
The SCT library uses the *chain-of-responsibility* design pattern for data processing. Each compression and decompression algorithm is implemented as an independent module, which can be freely reused for its task. The module chaining is performed by the `ChainBuilder` class, which creates, extends and ends the data flow chain with the help of the `ChainAdapter` class. `ChainAdapter` class implements the `Chainable` interface, which extends Java's `Consumer` interface. The `Consumer` interface is a functional interface representing an operation that accepts a single input argument and returns no result [24].

There are currently two ways of how a chainable module can be implemented. Either the whole class implements the `Chainable` interface, or one method from a class has such parameters that it can be chained using `ChainBuilder`. In the first case, the input data are passed via the overridden method `accept`, which accepts the given input data, processes them, and forwards them to the `Consumer`. The second case is more suitable if multiple input formats can be processed. There are multiple methods in a class that can be chained. All of the chainable methods from a class has to provide an interface such that the first parameter of the method is an input object, and the second parameter is the `Consumer`.

The primary requirement for a module to be added to the data flow chain using the `ChainBuilder` is to take the output from the preceding module as its input and provide an output that the following chain segment is able to process.

The UML class diagram for the `ChainBuilder`, the `ChainAdapter` and the `Consumer` is shown in Figure 2.1.

Figure 2.1: UML class diagram for the `ChainBuilder`, the `ChainAdapter` and the `Consumer`.



2.5 Input and output handling

Input and output of data to and from the library are handled by `FileIO` class.

For the compression, instances of this class need only one parameter - the size of a block. This parameter specifies how big the chunks of loaded data should be in bytes. `FileIO` reads chunks of bytes from the specified input file, wraps them into the `ByteBuffer` and sends them down the chain for further processing. While the input file is being processed, the compressed data chunks are stored in a temporary object in the memory in the form of

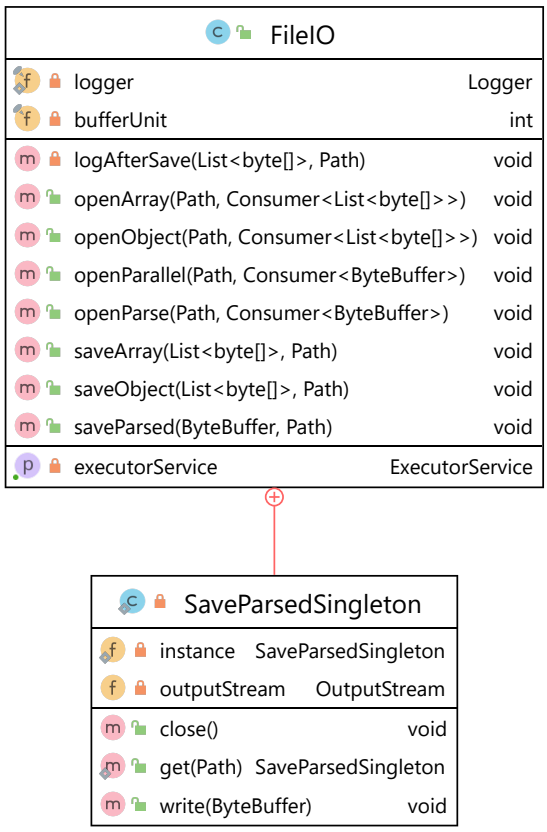
`List<byte[]>`. After the whole file is processed, this object is written in the output file at once.

For the decompression, instances of `FileIO` class do not need any additional information. All of the data are loaded from the compressed file into the memory in the form of `List<byte[]>`. This object is then passed down the chain for further processing. The first method in the chain parses this object into smaller `ByteBuffers` and decompresses them one by one. Decompressed data are stored in the output file as they are processed, using `SaveParsedSingleton` class. *Singleton* design pattern was chosen because it simplifies output stream handling and makes writing multiple chunks of data into the output file safer.

In conclusion, memory management is pretty ineffective. When compressing a file, the data are read in chunks, but they are not written in the output file as they are processed, which means unnecessary memory usage. Due to this design, the decompression also has to load the whole object into the memory and only then can it process the data in parts.

The UML class diagram for the `FileIO` is shown in Figure 2.2.

Figure 2.2: UML class diagram for the `FileIO`.



2.6 Implemented algorithms

This section focuses on already implemented compression methods. More information, including things such as pseudocodes and usage examples, can be found in the thesis associated with the particular compression algorithm as described in Section 2.2.

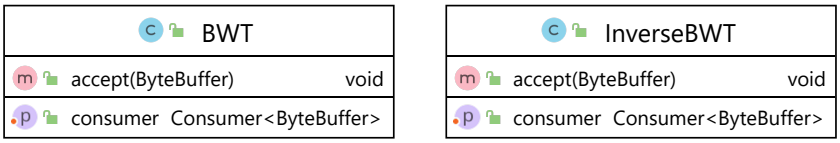
2.6.1 BWT

The Burrows-Wheeler transform (BWT from now on) is a compression method used to prepare data for use with data compression techniques such as the Huffman and the Arithmetic coding. It was invented and described by Michael Burrows and David Wheeler in 1994. It is based on a previously unpublished transformation discovered by Wheeler in 1983 [25].

The transform is done by creating a table of all the circular shifts of data bytes, followed by lexicographical sort and by extracting the last column and the index of the original data in the set of sorted permutations. The remarkable thing about the BWT is not that it generates a more easily encoded output, an ordinary sort would do that, but that it is reversible, allowing the original data to be reconstructed from the transformed data [17].

The implementation uses the `ByteBuffer` class both at the input and at the output. The UML class diagram for the `BWT` and the `InverseBWT` is shown in Figure 2.3.

Figure 2.3: UML class diagram for the BWT method.



2.6.2 MTF

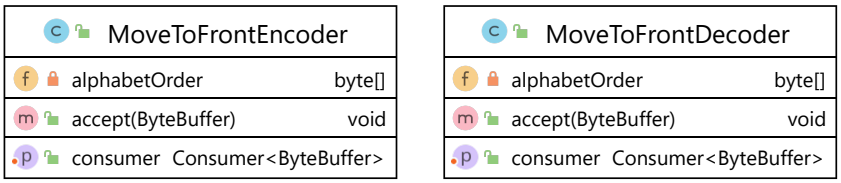
The Move-to-front transform (MTF for short) is a compression algorithm used to improve the performance of compression techniques by decreasing information entropy [26]. It doesn't compress the data by itself but instead transforms the input data to help following algorithms, such as RLE, with more efficient compression [8].

As the name suggests, this compression algorithm uses a list of possible symbols and modifies this list at every cycle (moving one symbol, the last used). Long sequences of identical symbols are replaced by as many zeros, whereas when a symbol has not been used in a long time, it is replaced with a large number. Thus at the end, the data is transformed into a sequence of integers. If the data shows a lot of local correlations, then these integers tend

to be small. This algorithm is designed to improve the performance of entropy encoding techniques [27, 17].

The implementation uses the `ByteBuffer` class both at the input and at the output. The UML class diagram for the `MoveToFrontEncoder` and the `MoveToFrontDecoder` is shown in Figure 2.4.

Figure 2.4: UML class diagram for the MTF method.



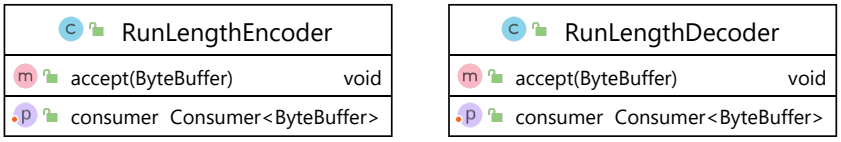
2.6.3 RLE

The Run-length encoding (RLE from now on) is a simple and widely used data compression algorithm that offers excellent compression ratios with the data that contain lots of redundant symbols [28]. The output of the MTF algorithm is a perfect example of such data with lower entropy and high redundancy and thus, making it suitable for use with the RLE algorithm [26].

This algorithm converts consecutive long sequences of identical symbol runs into a code consisting of the symbol and the number marking the length of the run. The longer the run, the better the compression ratio [29].

The implementation uses the `ByteBuffer` class both at the input and at the output. The UML class diagram for the `RunLengthEncoder` and the `RunLengthDecoder` is shown in Figure 2.5.

Figure 2.5: UML class diagram for the RLE method.



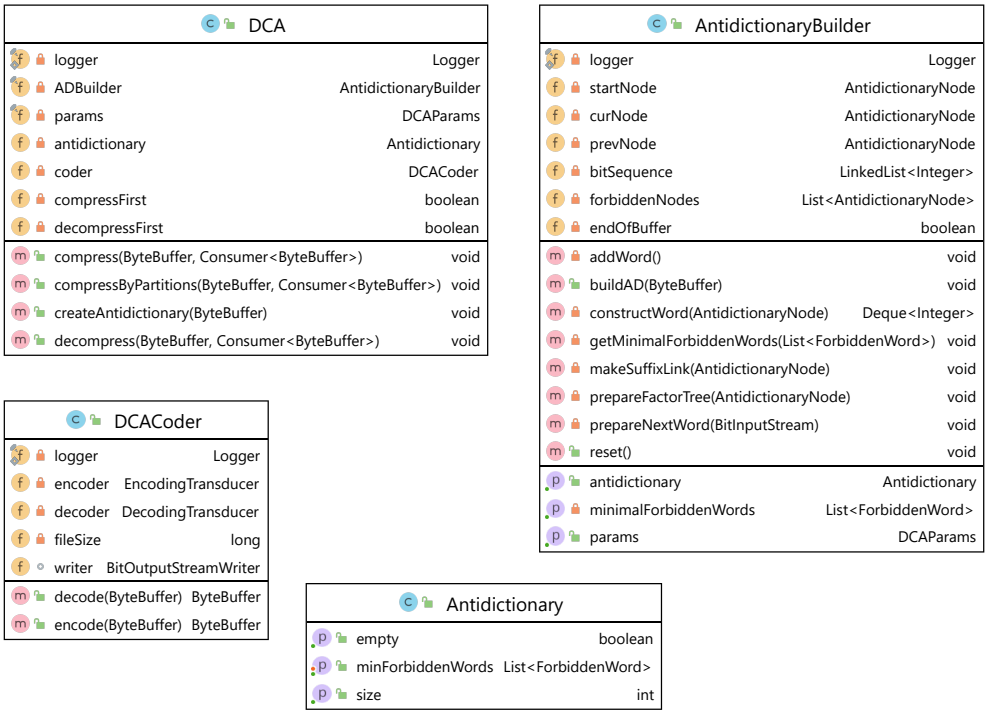
2.6.4 DCA

Data Compression using Antidictionaries (DCA for short) is a semi-adaptive context compression method using the binary alphabet. The coding of every symbol (a bit in this case) is dependant on the surrounding symbols, thus making the DCA a context method. During the compression, DCA uses the so-called antidictionary - a data structure that holds strings that do not occur in the input data. Since the antidictionary is specific for every input and needs to be built independently before the actual compression takes place, the DCA

needs two passes over the input data to compress them. The construction of the antidictionary is the most time and memory consuming operation of the DCA method. It is also important to note that the input specific antidictionary needs to be attached to the compressed data after the compression for the decompression to be possible [14].

The implementation uses the `ByteBuffer` class both at the input and at the output. The UML class diagram for the `DCA`, `DCACoder`, `Antidictionary` and `AntidictionaryBuilder` is shown in Figure 2.6.

Figure 2.6: UML class diagram for the DCA method.



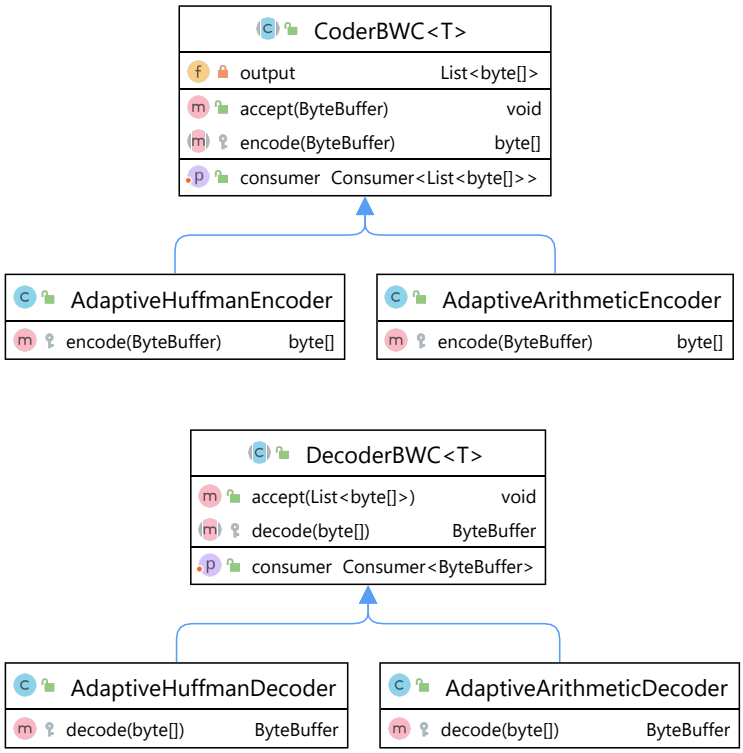
2.6.5 Adaptive entropy coders

Two adaptive arithmetic entropy coders are implemented in the SCT library. These two adaptive compression methods are Adaptive Arithmetic Coding (Adaptive AC from now on) and Adaptive Huffman Coding (Adaptive HC from now on). Each of them is closely described in dedicated subsections.

Both of the adaptive compression algorithms use `ByteBuffer` at the input, and `List<byte[]>` at the output. Their associated decompression algorithms use `List<byte[]>` at the input, and `ByteBuffer` at the output. Abstract classes `CoderBWC` and `DecoderBWC` serve as wrappers for both of these coders, and every new coder added to the SCT library has to implement these two abstract classes to be sufficient for further use in the library [17].

An open-source library developed by Nayuki outsources both Adaptive AC and Adaptive HC. [30, 31]. Classes `AdaptiveArithmeticEncoder`, `AdaptiveArithmeticDecoder`, `AdaptiveHuffmanEncoder` and `AdaptiveHuffmanDecoder`, implement already mentioned abstract classes `CoderBWC` and `DecoderBWC` and slightly adjust data before forwarding them to the external library for further processing [17]. The associated UML class diagram for both entropy coders is shown in Figure 2.7.

Figure 2.7: UML class diagram for the adaptive entropy coders.



2.6.5.1 Adaptive Arithmetic coding

Arithmetic coding (AC for short) is a form of entropy encoding used in lossless data compression that allows each symbol to be coded with a non-whole number of bits (when averaged over the entire data), thus improving the compression ratio. Arithmetic coding represents the current data as a range, defined by lower and upper bounds and encodes the whole data into a single number from this interval, an arbitrary-precision fraction n where $0 \leq n < 1$. Starting with the interval $\langle 1, 0)$, each interval is divided into several subintervals, which sizes are proportional to the current probability of the corresponding symbols of the alphabet. The subinterval from the currently coded symbol is

then taken as the interval for the next symbol. The output is a number from the interval of the last symbol [32].

Adaptive AC is an adaptive compression method that starts with flat probabilities of symbols and updates them after each symbol is processed, thus making it reflect the statistics of the data being compressed [30]. When the coder encodes the data, it counts the frequencies of the symbols that have occurred so far in order to obtain a model of the probabilities for the future symbols. The coder is called adaptive because the model evolves gradually while the coder processes its input [17].

2.6.5.2 Adaptive Huffman coding

Huffman coding (HC for short) is a form of entropy encoding based on the probabilities of the symbols occurring in the alphabet from the input string. HC encodes symbols into variable lengths, depending on the probability of each symbol. The symbols which appear more frequently occupy fewer bits than symbols with less frequency. First, the algorithm counts frequencies and probabilities of all individual symbols from the alphabet in the input string. According to these frequencies, it then generates a binary tree, which has edges with values 0 or 1, and in its leaf nodes, there are symbols from the input alphabet. It follows the principle that the higher the probability of symbol, the nearer is the appearance of a node to the root of the tree. Then these symbols are replaced with binary code, which corresponds to the concatenation of values of edges, which are passed on the way from the root to the desired leaf node. Since one of the values 0 or 1 always appears on edges leading to the right successor (the other to the left successor) and symbols from the input alphabet are located only in leaf nodes, the resulting code is a prefix. It is necessary to attach this tree or at least the information about frequencies of appearance of single symbols in input data to the resulting output binary sequence. Then the decompression is possible [8].

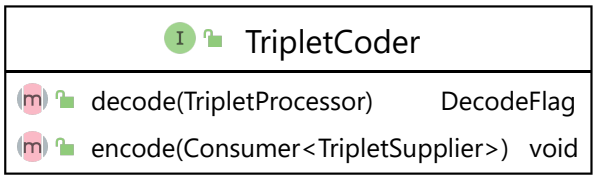
Adaptive HC is an adaptive compression method that calculates the probabilities dynamically based on recent actual frequencies in the sequence of symbols from the input alphabet and changes the coding tree structure to match the updated probability estimates. The coder is called adaptive because the binary tree is changing simultaneously with the processed input data in order for the coder to remain optimal for the current probability estimates. Since it permits building the code as the symbols are being processed, having no initial knowledge of source distribution, only one pass over the data is required. Another benefit of the one-pass procedure is that the source can be encoded in real-time [8, 17].

2.6.6 Triplet coders

Triplet coders are modules that produce triplets from the given input data during the compression and accept triplets and convert them to the original data during the decompression. Triplet coders are always paired with triplet processors that accumulate and encode triplets during the compression and decode the given data into triplets during the decompression.

Each triplet coder must implement the `TripletCoder` interface. During the compression, each implementation accepts the `ByteBuffer` class at the input and produces triplets at the output. During the decompression, it is the other way around. The UML class diagram for the `TripletCoder` is shown in Figure 2.8.

Figure 2.8: UML class diagram for the `TripletCoder`.



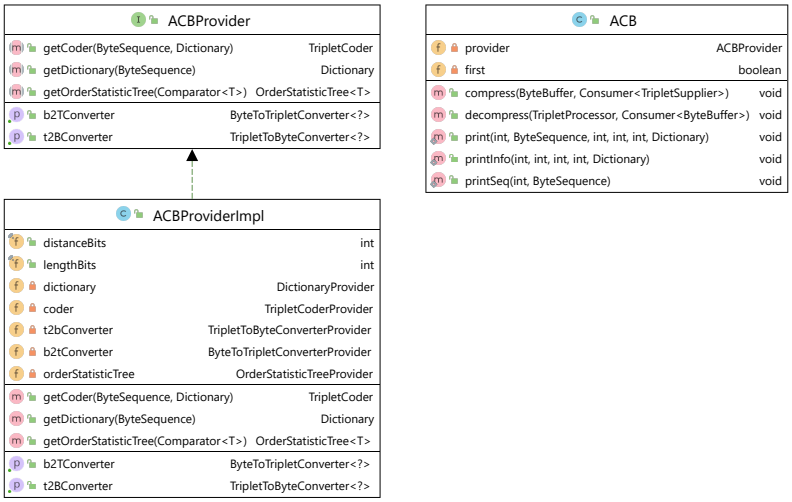
2.6.6.1 ACB

Associative Coder of Buyanovsky (ACB from now on) is an adaptive context dictionary compression method that substitutes substrings of the string to be processed with fixed-length indices into a dictionary. The indices, of course, must be shorter than the substrings it replaces for compression to occur. The dictionary is not attached to the output of the ACB compression but is built in the same way during the decompression [33].

During the compression, the current context is compared with items in the dictionary (with their context part, comparison from right to left) and the actual content is compared with contents in the dictionary (comparison from left to right). The ACB then produces triplet (d, cnt, s), where d is the distance between the best context and the best content, cnt is a number of matched symbols, and s is the first unmatched symbol in the look-ahead buffer. The dictionary is then updated, and the look-ahead buffer is shifted $cnt + 1$ positions to the right [8].

The UML class diagram for the ACB implementation is shown in Figure 2.9.

Figure 2.9: UML class diagram for the ACB method.



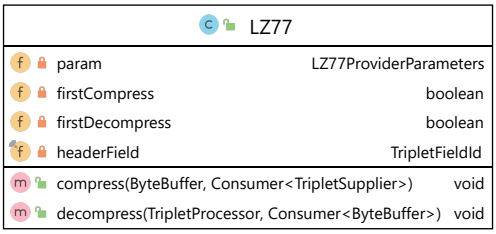
2.6.6.2 LZ77

LZ77 is an adaptive dictionary compression method, first introduced in 1977 by Abraham Lempel and Jacob Ziv. Method's name is derived from the first letters of the authors' surnames and the year it was first introduced. It uses a sliding window divided into search buffer and look-ahead buffer. During the compression, the sliding window moves from the left to the right of the input data, and its size is crucial for the method's effectiveness. Due to its significantly faster decompression than compression, LZ77 is suitable for usage in situations when the data are compressed once and then decompressed many times over [8, 15].

First, the longest prefix of a look-ahead buffer that starts in the search buffer is found. This prefix is then encoded as triplet (i, j, s) where i is the distance of the beginning of the found prefix from the end of the search buffer, j is the length of the found prefix, and s is the first symbol after the prefix in the look-ahead buffer. The number of bits written to the output data depends on the used encoding of numbers [8].

The UML class diagram for the LZ77 implementation is shown in Figure 2.10.

Figure 2.10: UML class diagram for the LZ77 method.



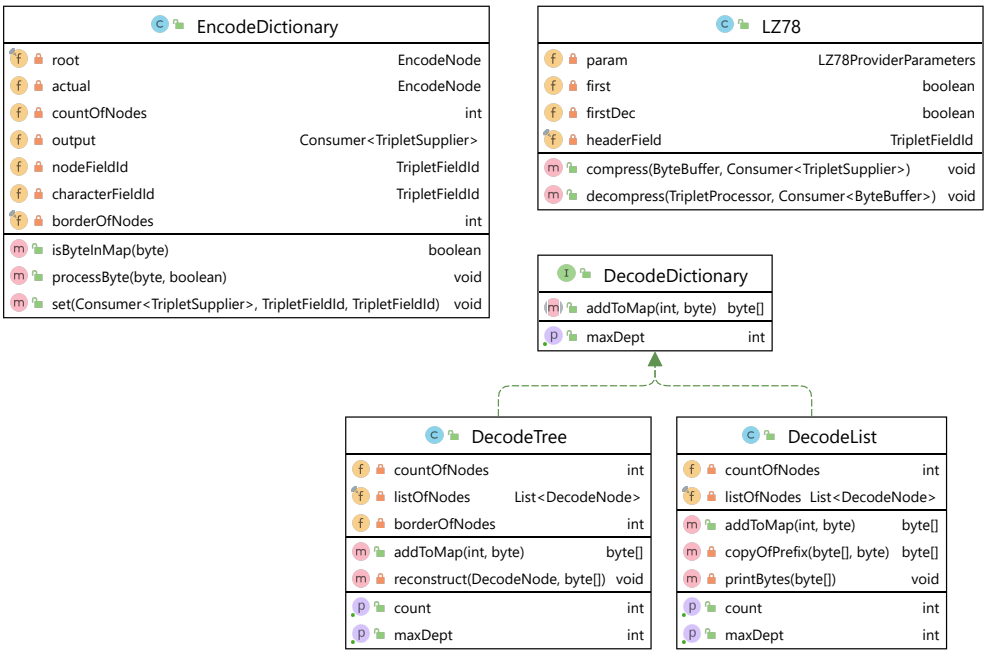
2.6.6.3 LZ78

LZ78 is an adaptive dictionary compression method, first introduced in 1978 by Abraham Lempel and Jacob Ziv. Same as with the LZ77, the method's name is derived from the first letters of the authors' surnames and the year it was first introduced. LZ78 works by entering phrases into a dictionary and then outputting the dictionary index instead of the phrase when a repeated occurrence of that particular phrase is found. Every step LZ78 will send a triplet (i, s) to the output, where i is an index of the phrase into the dictionary, and s is the next symbol following immediately after the found phrase. The dictionary is represented like the trie with numbered nodes. If we go from the root to a particular node, we will get the phrase from the input text. In each step, LZ78 looks for the longest phrase in the dictionary corresponding to the unprocessed part of the input data. Index of this phrase, together with the symbol, which follows the found part in the input data, are then sent to the output. The old phrase extended by the new symbol is then put into the dictionary. This new phrase is numbered by the smallest possible number. The coding will start with a tree that has only one node, which represents an empty string [8].

Unlike LZ77, the time complexity for the compression and the decompression is asymptotically the same, meaning it is fitting for usage when the compression and the decompression happen about as often [15].

The UML class diagram for the LZ78 implementation is shown in Figure 2.11.

Figure 2.11: UML class diagram for the LZ78 method.



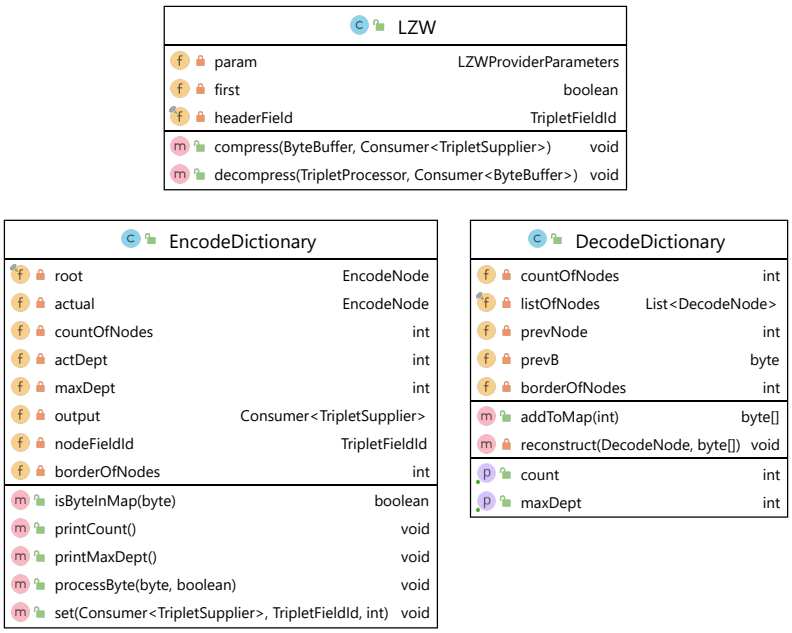
2.6.6.4 LZW

LZW is an adaptive dictionary compression method published by Terry Welch in 1984 as an improved modification of the LZ78 method published by Lempel and Ziv in 1978. Its name is derived from the first letters of the authors' surnames (Lempel, Ziv, Welch). The LZW method outputs triplet (i), where i is an index of the phrase into the dictionary and, unlike the LZ78 method, does not need to include the next symbol following immediately after the found phrase in the output triplet. This, however, causes rapid growth of the count of entries in the dictionary [15].

The LZW method provided a better compression ratio in most applications than any well-known method available up to that time and, due to this advantage, it became the first widely used universal data compression method on computers. It is used in PDF and GIF file formats [8].

The UML class diagram for the LZW implementation is shown in Figure 2.12.

Figure 2.12: UML class diagram for the LZW method.



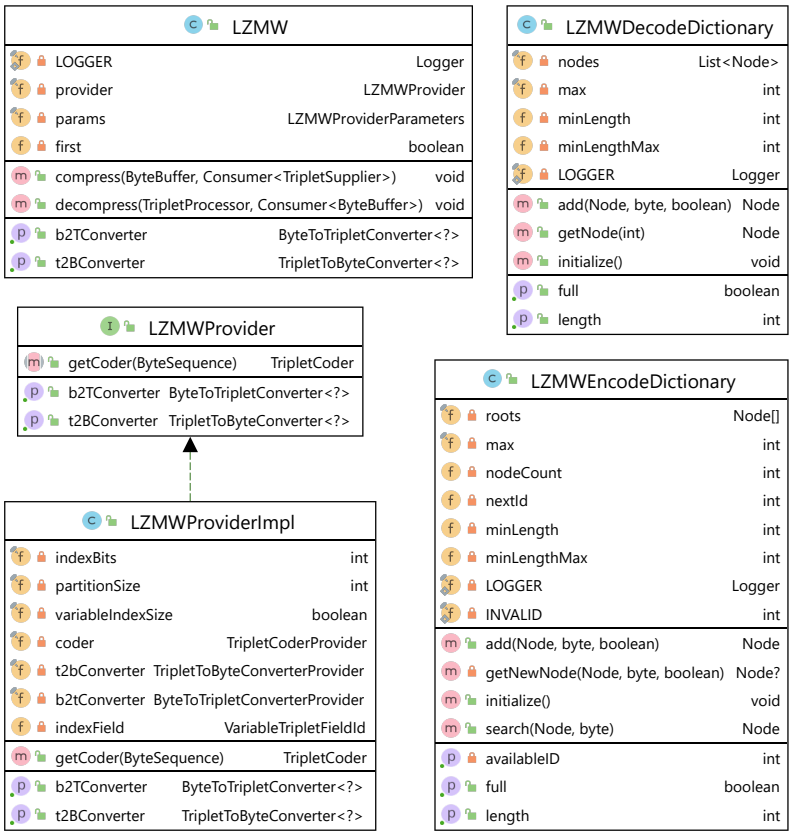
2.6.6.5 LZMW

LZMW is an adaptive dictionary compression method published by Victor Miller and Mark Wegman in 1985 as an improved modification of the LZW method. Its name is derived from the first letters of the authors' surnames (Lempel, Ziv, Miller, Wegman). Like the LZW method, the LZMW outputs triplet (i), where i is an index of the phrase into the dictionary [16]. The

LZW's problem is that it is slow to adapt to its input since phrases in the dictionary become only one symbol longer at a time. The LZMW method overcomes this problem with the principle that it searches the input for the longest string already in the dictionary (the "current" match) and adds the concatenation of the previous match with the current match to the dictionary. Thus, the dictionary phrases can grow by more than one symbol at a time. This implies that the LZMW's dictionary generally adapts to the input faster than the LZW's dictionary, usually providing better compression ratios [9].

The UML class diagram for the LZMW implementation is shown in Figure 2.13.

Figure 2.13: UML class diagram for the LZMW method.



2.6.6 LZAP

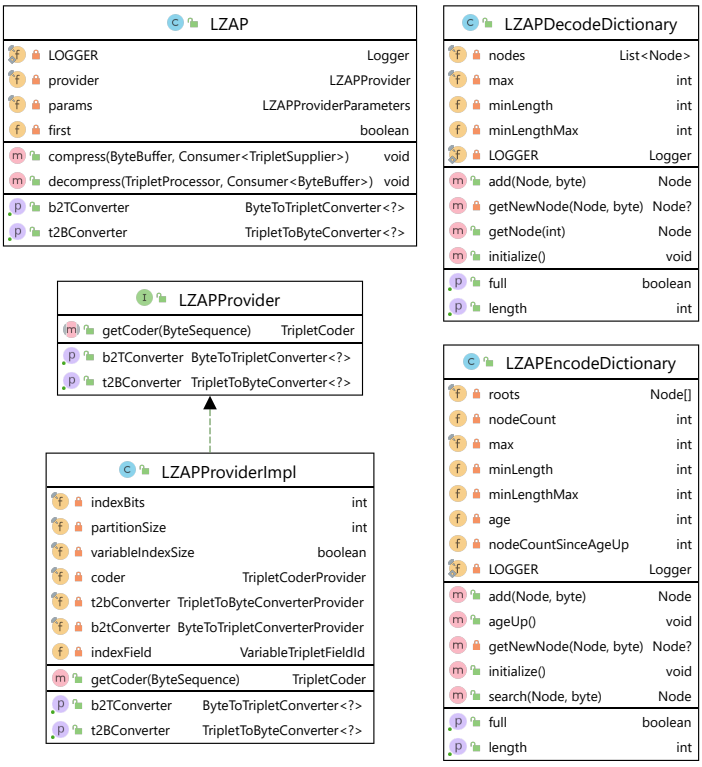
LZAP is an adaptive dictionary compression method published by James Storer in 1988 as an improved modification of the LZMW method. AP in the method's name stands for "All Prefixes". Instead of adding just the concatenation of the previous match with the current match to the dictionary,

as the LZMW does, add the concatenations of the previous match with each initial substring of the current match [34].

The LZAP method adapts to its input fast, like the LZMW method does, but eliminates the need for backtracking, a feature that makes it faster than LZMW. The LZAP adds more phrases to its dictionary than the LZMW does, so it takes more bits to represent the position of a phrase. At the same time, the LZAP provides a more extensive selection of dictionary phrases as matches for the input string, so it ends up compressing slightly better than the LZMW while being faster (because of the simpler dictionary data structure, which eliminates the need for backtracking) [9]. Like the LZW and the LZMW methods, the LZAP method outputs triplet (i), where i is an index of the phrase into the dictionary [16].

The UML class diagram for the LZAP implementation is shown in Figure 2.14.

Figure 2.14: UML class diagram for the LZAP method.



2.6.6.7 LZY

LZY is an adaptive dictionary compression method published by Dan Bernstein, and it is an improved modification of the LZAP method. Y in the method's name stands for "Yabba", which came from the input string origi-

nally used to test the algorithm. The LZW and the LZAP methods preserved the dictionary's structure in the state that if any phrase is chosen at any step during the algorithm, it is also possible to find all phrases' prefixes in the dictionary. The LZY method follows the same principle but also ensures that all of the phrases' suffixes can be found in the dictionary. Like the LZW, the LZMW and the LZAP methods, the LZY method outputs triplet (i), where i is an index of the phrase into the dictionary [16].

The UML class diagram for the LZY implementation is shown in Figure 2.15.

Figure 2.15: UML class diagram for the LZY method.

2.6.6.8 LZFSE

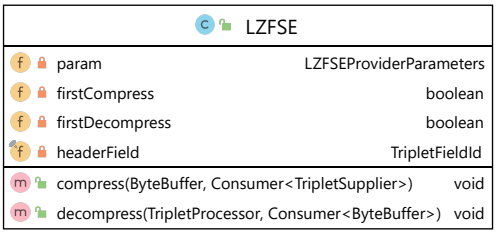
LZFSE is a compression method developed by Apple and first published in 2015. It is composed of two parts, the frontend part and the backend part. The frontend part produces triplets, while the backend part encodes the triplets themselves. From now on, in the scope of this thesis, the LSFSE method is

understood as the frontend part of the original method published by Apple, while the backend part is referred to as the Finite State Entropy triplet processor [18]. This naming convention was proposed by the method's implementer into the SCT library, Ladislav Zemek, simply because the frontend part of the LZFSE method does not have an official dedicated name, while the backend part can be referred to by its functionality - Finite State Entropy encoding triplet processor, which will be discussed in the next section dedicated to the triplet processors. So, in conclusion, the LZFSE method and Finite State Entropy encoding triplet processor from the scope of this thesis combined give us the LZFSE method developed by Apple.

The LZFSE is a method similar in functionality to the LZ77. It also operates with a sliding window divided into search buffer and look-ahead buffer. Compared to the LZ77 method, the main difference is that if it fails to find the prefix longer than the method's parameter K, the LZFSE then skips this prefix and remembers how big the skipped part was. If a sufficiently long prefix is subsequently found, the method produces a triplet (l, s, i, j), where l is the length of s, s is a string of skipped symbols, i is the length from the beginning of the search buffer to the beginning of the prefix and j is the length of the prefix [18].

The UML class diagram for the LZFSE implementation is shown in Figure 2.16.

Figure 2.16: UML class diagram for the LZFSE method.



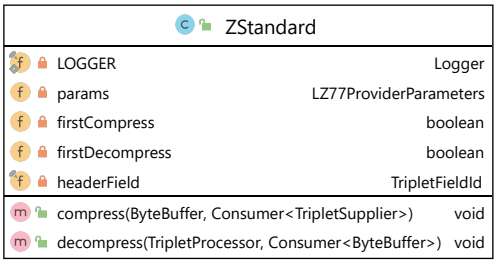
2.6.6.9 Zstandard

Zstandard is a compression method developed by Yann Collet at Facebook and published in 2015. Same as the LZFSE method, it is composed of two parts, the frontend part and the backend part. The frontend part is represented by the LZ77 method, and the Finite State Entropy triplet processor represents the backend part [18].

In the scope of this thesis, the LZ77 and the Zstandard triplet coders have the same functionality, and the Zstandard triplet coder differs in only a few lines of code, such as the logger messages.

The UML class diagram for the Zstandard implementation is shown in Figure 2.17.

Figure 2.17: UML class diagram for the Zstandard method.

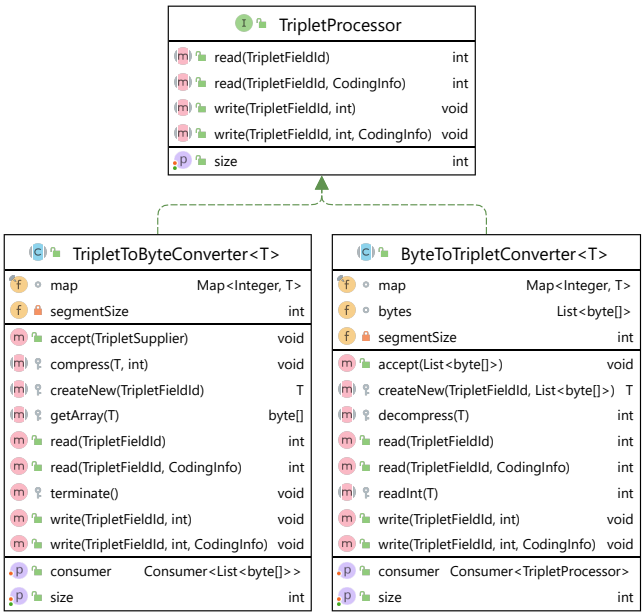


2.6.7 Triplet processors

Triplet processors are modules that accept triplets passed from the triplet coders during the compression and encode them. During the decompression, they decode the given data into triplets and pass them to the triplet coders.

Each triplet processor is split into two parts, the encoder, which is used during the compression, and the decoder, which is used during the decompression. Every encoder extends the abstract class `TripletToByteConverter`, and every decoder extends the abstract class `ByteToTripletConverter`. Both of these abstract classes implement the `TripletProcessor` interface. During the compression, each triplet processor accepts triplets at the input and produces the `ByteBuffer` class at the output. During the decompression, it is the other way around. The UML class diagram for the `TripletToByteConverter`, `ByteToTripletConverter` and `TripletProcessor` is shown in Figure 2.18.

Figure 2.18: UML class diagram for the `TripletToByteConverter`, `ByteToTripletConverter` and `TripletProcessor`.

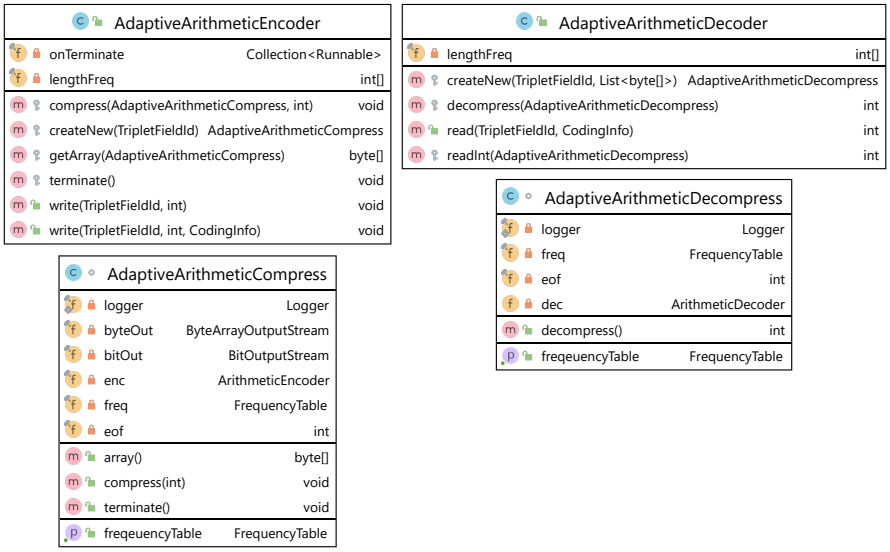


2.6.7.1 Adaptive Arithmetic coding

Adaptive Arithmetic coding triplet processor utilises the same algorithm as is used in the triplet coder Adaptive Arithmetic coding. Also, the same open-source library developed by Nayuki is used in both the triplet coder and the triplet processor [5, 30]. While the Adaptive Arithmetic coding triplet coder, discussed in Subsection 2.6.5.1, encodes the input data as a whole, its triplet processor counterpart chooses a different approach. It creates a separate coder instance for every element of triplet, allowing Adaptive Arithmetic coding triplet processor to have a separated frequency table for each alphabet, thus achieving better compression [5].

During the compression, the `AdaptiveArithmeticEncoder` class encodes the triplet's elements, while the `AdaptiveArithmeticDecoder` class is used during the decompression. The UML class diagram for the Adaptive Arithmetic coding triplet processor is shown in Figure 2.19.

Figure 2.19: UML class diagram for the Adaptive Arithmetic coding.



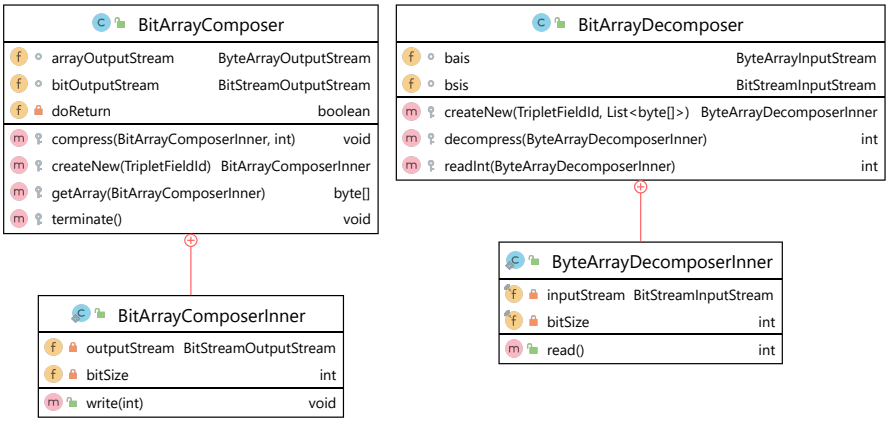
2.6.7.2 Bit Array Composing

Bit Array Composing takes the triplet value and stores it into a byte array, buffering and aligning bits. It does not compress the given triplets and simply accumulates them behind each other into the output byte array. This triplet processor was implemented for comparison with Adaptive Arithmetic coding, to see how much more efficient it is than using no coding at all [5].

During the compression, the `BitArrayComposer` class encodes the triplet's elements, while the `BitArrayDecomposer` class is used during the decompres-

sion. The UML class diagram for the Bit Array Composing triplet processor is shown in Figure 2.20.

Figure 2.20: UML class diagram for the Bit Array Composing.



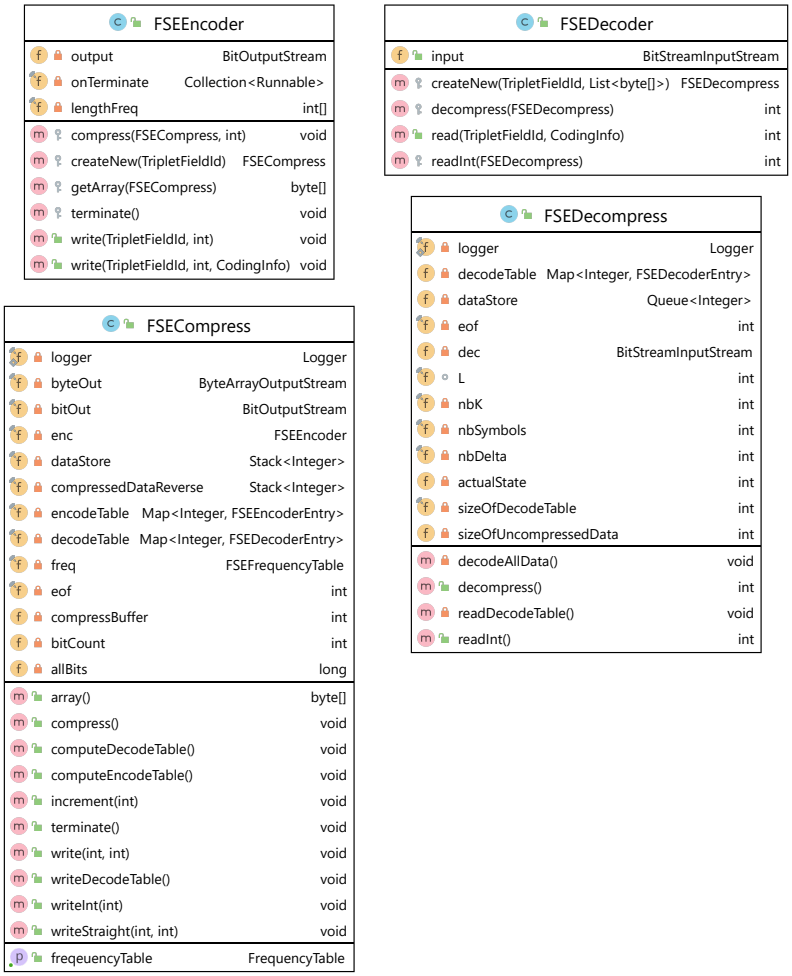
2.6.7.3 Finite State Entropy coding

Finite State Entropy (FSE from now on) coding triplet processor is a member of the family of Asymmetric numeral systems (ANS for short) entropy encodings introduced by Jarosław Duda. ANS combines the compression ratio of arithmetic coding (which uses a nearly accurate probability distribution) with a processing cost similar to that of Huffman coding [18].

The basic idea of ANS is to encode information into a single natural number, x. In the standard binary number system, we can add a bit s of information to x by appending s at the end of x, which gives us $x' = 2x + s$. For an entropy coder, this is optimal if $P(0) = P(1) = 1/2$. ANS generalizes this process for arbitrary sets of symbols with an accompanying probability distribution. Instead of using multiplication, an operation quite costly on processing resources, Finite State Entropy coding triplet processor constructs a finite-state machine to operate on a large alphabet, thus achieving excellent compression ratio and compression and decompression times. The finite-state machine used by the FSE is often represented as a table, so the FSE is sometimes called table ANS (tANS) [35].

During the compression, the `FSEEncoder` class encodes the triplet's elements, while the `FSEDecoder` class is used during the decompression. The UML class diagram for the Finite State Entropy coding triplet processor is shown in Figure 2.21.

Figure 2.21: UML class diagram for the Finite State Entropy coding.



2.7 Operational status of the implemented modules

During the library's lifetime, countless new additions have been implemented. A lot of them have not only added new modules but also modified the existing ones. These modifications have often not been sufficiently tested due to neglected tests, and some of them have quietly brought errors into the codebase. The following list lists discovered non-functioning modules in the SCT library.

- **ACB -** The use of the LZW module can sometimes result in an error. The method functions as expected if used on regular plain text files but fails if used on more complex data, such as a picture or an audio file.
- **DCA -** The use of the DCA module results in an infinite loop during the decompression.

- **LZAP -** The use of the LZAP module results in an error if the input file is bigger than the specified size of the working buffer. This means that the LZAP module works correctly only if it can process the whole input file in one chunk.
- **LZW -** The use of the LZW module results in an error during the decompression.

2.8 Non-functional requirements

Non-functional requirements refer to constraints and behavioural properties of the system. They specify criteria that can be used to judge the operation of a system rather than specific behaviours. A typical non-functional requirement contains a unique name and number and a brief summary. This information is used to understand better why the requirement is needed and can be used to track the requirement through the development of the system.

- **N1 SCT library -** Implementation has to be done as a part of the Small Compression Toolkit library.
- **N2 Java programming language -** Implementation has to be done in Java programming language which is the programming language used to develop SCT library.
- **N3 Documentation -** Implementation has to be appropriately and sufficiently documented, and this documentation has to be compliant with documentation standards for Java programming language.
- **N4 Readability -** The created source code has to be easily readable and has to follow Java programming language conventions.
- **N5 Extensibility -** Implementation has to be done in a way that future extensions could easily be developed and utilised in systems without losing existing capabilities.
- **N6 Testability -** Implementation has to be done in a way that it can be easily tested so the chance of finding faults in the library is high.

2.9 Functional requirements

Functional requirements specify a function that a system or system component must be able to perform. They refer to services that the system should provide. A typical functional requirement contains a unique name and number and a brief summary. This information is used to understand better why the requirement is needed and can be used to track the requirement through the development of the system.

- **F1 Unified module interface -** Implement a unified module interface that will be used by all of the implemented modules in the SCT library. The unified module interface shall allow random and free chaining of the implemented modules without any constraints.
- **F2 File format -** Implement a unified file format that will be used for the whole SCT library, and all of the implemented modules will work with it. The file format has to be easily expandable to support future additions to the library.
- **F3 Modular framework -** Implement a new modular framework for the SCT library that will use the new unified module interface and save a processed file under the unified file format.
- **F4 Client -** Implement a client which can be used to run the application from the command-line interface. The functionality of this client can be controlled by parameters or a configuration file.
- **F5 Tests -** Implement automated tests which can be run in order to assess the functionality and reliability of the library.
- **F6 Measurement tool -** Implement a measurement tool that can be used to compare compression algorithms on the same input. Input can be a file or a folder, and in the case of a folder, all subfolders shall be processed recursively. The output of this measurement tool shall be saved in the standard and familiar file format for better future processing of the results.

Chapter 3

Design and implementation

This chapter is dedicated to the implementation of required functionality, with particular emphasis given to meet functional and non-functional requirements. The assignment of this thesis was to redesign and implement the SCT library's modular framework and take into account the current and possible future modules. This chapter deals with this process and explains the steps needed to fulfil the assignment.

The first section is dedicated to designing and implementing the unified module interface. The second section shows the design and integration of the new unified interface into the SCT library and the implementation of the new modular framework. The third section deals with the implementation of the unified file format for the files produced by the SCT library. The fourth section is dedicated to implementing the command-line client for the SCT library together with a run configuration file for this client. The penultimate section introduces tests to the SCT library, and the last section shorty introduces the new measurement tool that will be used to measure compression algorithms' performance.

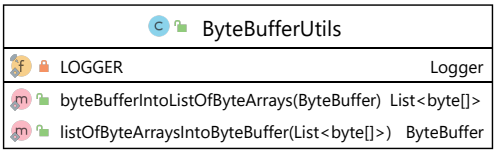
3.1 Unified module interface

The crucial task when designing the unified interface for the modules is to make sure that the interface is simple to understand, easy to use and that it makes sense in the context of the library. The SCT library works with raw data read from files as bytes. As shown in the Chapter 2 of this thesis, there are currently three different interfaces between modules - triplets, `ByteBuffer` and `List<byte[]>`.

First of all, let's deal with the `List<byte[]>`. The list of byte arrays is used as the output of the `TripletToByteConverter` class, which is used during the compression to encode triplets, and the input of the `ByteToTripletConverter` class, which is used during the decompression to decode data into triplets. This is by triplet's design and can not be changed. The easiest and most

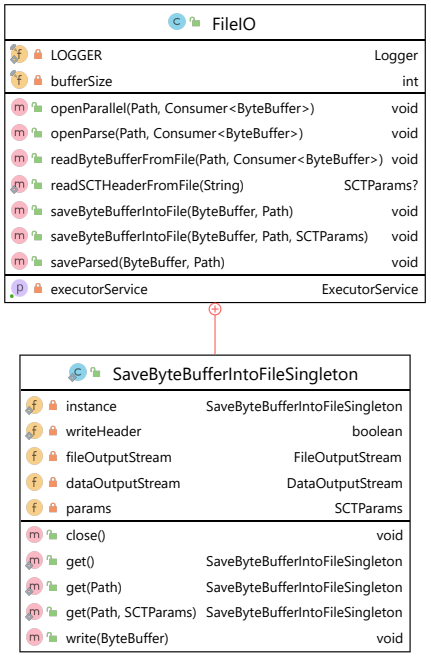
straightforward way of how to get rid of the `List<byte[]>` is to implement a simplistic converter. This converter is implemented in `ByteBufferUtils` class and is called after the triplets are encoded during the compression or before the decoding of the triplets takes place during the decompression. The UML class diagram for the `ByteBufferUtils` class is shown in Figure 3.1.

Figure 3.1: UML class diagram for the `ByteBufferUtils`.



Since the lists of byte arrays are not used by the `TripletToByteConverter` and `ByteToTripletConverter` classes anymore, there is only one place left where they are used - `FileIO` class. The author of this class decided to use them because, at the time of the initial implementation of the `FileIO` class, there was no need for a more robust solution. However, during the SCT library's lifecycle, many new extensions have been added to the library, and the authors of those additions chose the easiest way of integrating to the library - modifying interfaces of their modules to match those used by the `FileIO` class, instead of completely redesigning the `FileIO` class. This technical debt can be finally taken care of, and the UML class diagram for the reworked `FileIO` class is shown in Figure 3.2.

Figure 3.2: UML class diagram for the reworked `FileIO`.

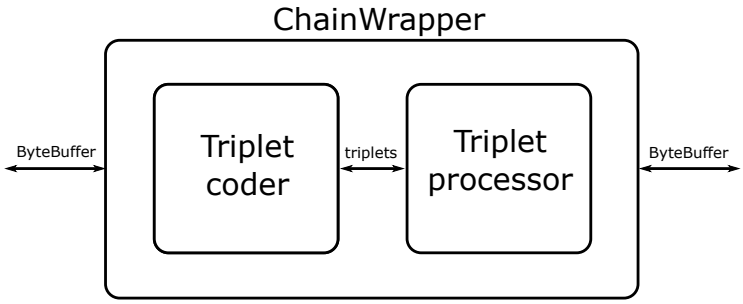


This change leaves only triplets and the `ByteBuffer` class as modules' interfaces. Since triplets are exclusively used only between triplet coders and triplet processors, the simple wrapper can wrap these two module types and provide a unified module interface to the outside environment. This wrapper is implemented in the `ChainWrapper` class and is presented in the following subsection. This last necessary change leaves only one last interface type in the SCT library - the `ByteBuffer` class, and thus the unified module interface is finally achieved. Due to its simplicity, the `ByteBuffer` class is an excellent solution for the foreseeable future. It wraps raw bytes into a Java class and provides useful operations upon them.

3.1.1 ChainWrapper

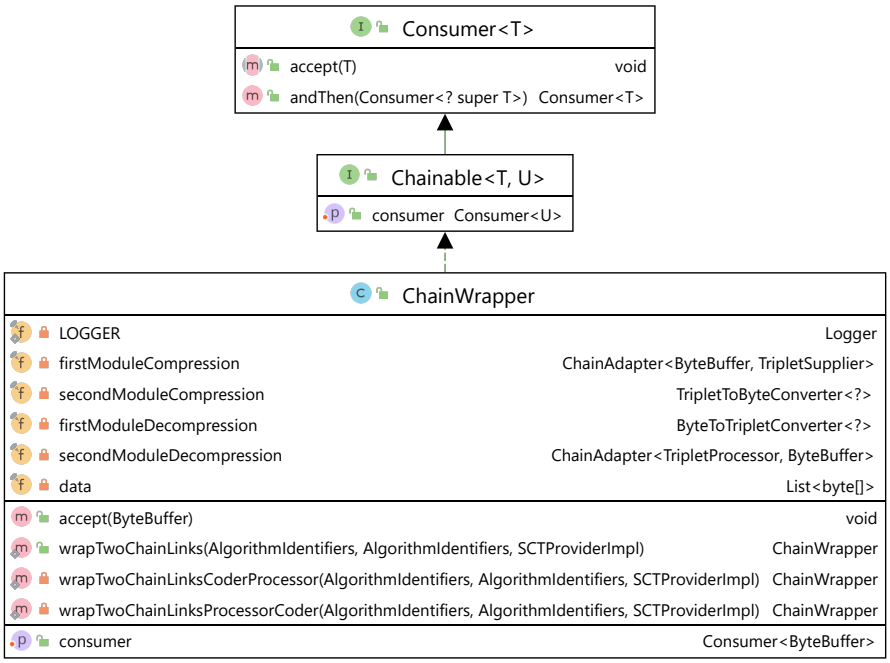
`ChainWrapper`, the same as any other module which wants to be part of the processing chain, implements the `Chainable` interface. This module encapsulates two succeeding chain links that exchange triplets between them, the triplet coder and the triplet processor, and provides a unified interface that operates with the `ByteBuffer` class. `ChainWrapper's` design is illustrated in Figure 3.3.

Figure 3.3: `ChainWrapper's` design.



`ChainUtils` class, responsible for adding new links to the chain and discussed later, recognises that two succeeding chain links exchange triplets and calls the `wrapTwoChainLinks` method, which returns desired instance of the `ChainWrapper` class. This instance is then added to the chain in the same way as any other module from the SCT library. During the compression and the decompression, the `ChainWrapper` forwards data to the two modules encapsulated in the `ChainWrapper`. The `ChainWrapper` class is implemented with an emphasis on its extensibility, and if any new triplet coder or processor is added to the SCT library, very minor and simple adaptations have to be made to its implementation. The UML class diagram for the `ChainWrapper` is shown in Figure 3.4.

Figure 3.4: UML class diagram for the `ChainWrapper`.



3.2 Modular framework

All modules now have the same interface at the input and the output. However, the SCT library does not currently have any framework that would allow free chaining of all modules. Up to now, every new extension to the library was implemented in a separate package, and every one of them has an individual client, parameter class and provider class. This approach is unsustainable and highly limits the modules' usability and extensibility.

The newly implemented modular framework solves all of the limitations of the old solution and is designed to ease adding future extensions to the library. The following few subsections of this section are dedicated to describing all of the essential classes that together make up the new modular framework that is supposed to bring robustness and clarity to the SCT library.

3.2.1 SCTConfig

This class contains all necessary enumerations needed for the correct configuration of the SCT library for the compression and decompression needs. The `SCTConfig` class also specifies the extensions for the files compressed and decompressed by the SCT library.

Enumeration `AlgorithmIdentifiers` defines constants that are used to identify compression methods within the modular framework itself. Every new compression method to the library must be added to this enumeration.

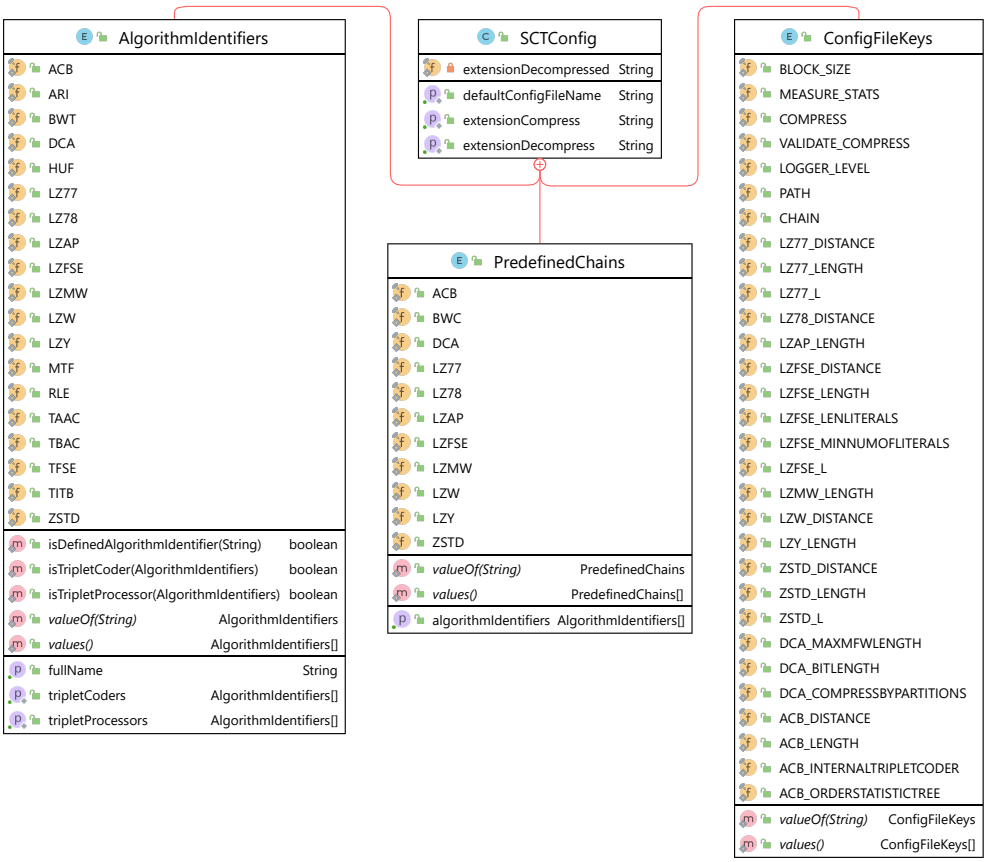
If the new addition is either a triplet coder or triplet processor, then it must be appropriately marked as such in `AlgorithmIdentifiers` enumeration.

Enumeration `PredefinedChains` contains processing chains that were part of the library even before of the implementation of the new modular framework. This enumeration exists purely for legacy reasons, and every one of these predefined chains covers one compression method implementation that was part of one of the library's associated theses. This enumeration comes in handy when comparing different compression methods that are part of this library on the same input data, such as corpora.

The new modular framework can also be configured with the configuration file. File `sct_default.conf` contains default configuration, such as the size of the block. Each constant from enumeration `ConfigFileKeys` from the `SCTConfig` class specifies keys that can be used to configure compression methods from the library.

The UML class diagram for the `SCTConfig` class is shown in Figure 3.5.

Figure 3.5: UML class diagram for the `SCTConfig`.

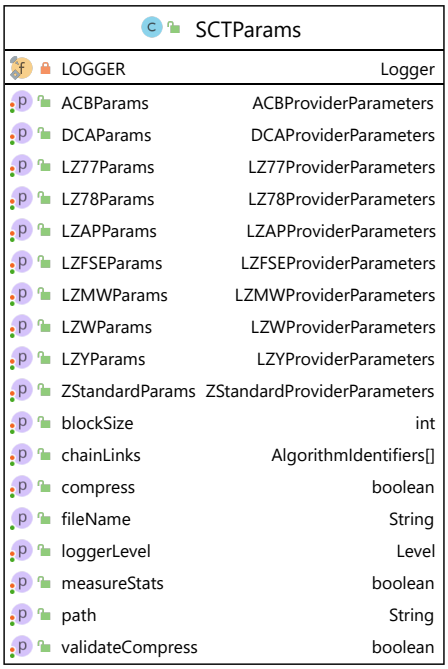


3.2.2 SCTParams

This class holds all parameters necessary for the correct functionality of the modular framework, such as for creating an instance of the SCTProvider class. These parameters are set by the user via command-line interface or config file and used when compressing and decompressing files. Since the SCTParams class aggregates all of the essential variables used by the compression methods during the compression, it is attached to the compressed file as a header that is later parsed and used during the decompression.

The UML class diagram for the `SCTParams` class is shown in Figure 3.6.

Figure 3.6: UML class diagram for the `SCTParams`.

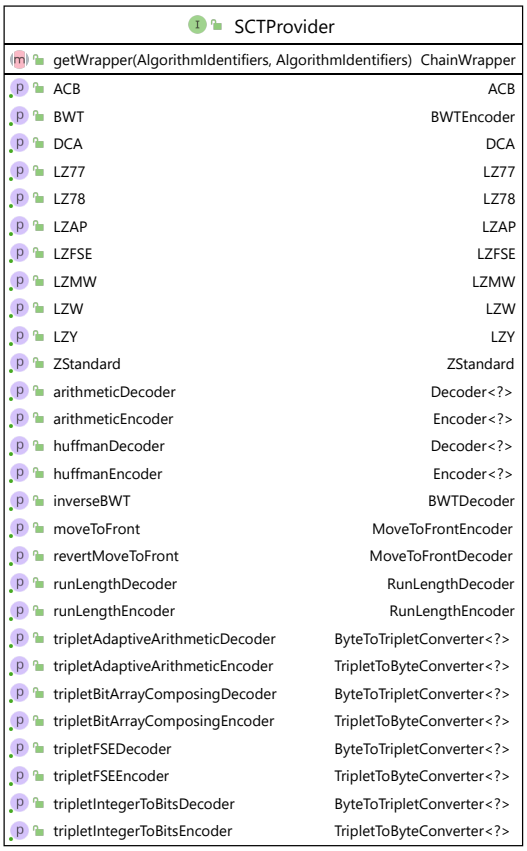


3.2.3 SCTProvider

The idea behind this component is to decouple initialization logic from execution logic, so the compression and the decompression run solely with configured modules and are not delayed by initialization during the runtime. `SCTProvider` interface is a form of data provider that accepts specified run configuration in the form of the `SCTParams` instance, initializes modules' constructors using these parameters, and provides them under a unified interface. Class `SCTProviderImpl` implements the `SCTProvider` interface and is instantiated using the passed parameters during the compression and the decompression.

The UML class diagram for the `SCTProvider` interface is shown in Figure 3.7.

Figure 3.7: UML class diagram for the `SCTProvider`.

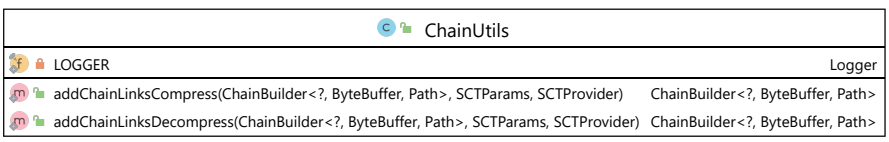


3.2.4 ChainUtils

`ChainUtils` class provides essential methods used to build compression and decompression chains from given parameters using the `SCTProvider`. An instance of the `SCTParamaters` class, passed to the `addChainLinksCompress` and `addChainLinksDecompress` methods, holds desired chain configuration that specifies what modules shall be linked to the chain.

The UML class diagram for the `ChainUtils` class is shown in Figure 3.8.

Figure 3.8: UML class diagram for the `ChainUtils`.



3.2.5 SCTUtils

This class aggregates important utilities used by the new modular framework in the SCT library. The most essentials are `compress`, `decompress` and `parseConfigFromConfigFile` class methods.

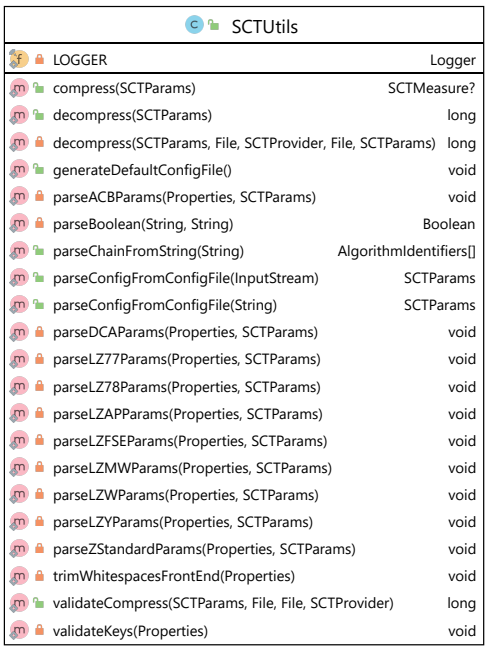
The `parseConfigFromConfigFile` method from the `SCTUtils` class is responsible for parsing the run configuration from the given configuration file. It firstly verifies if the passed configuration file is valid. Then, common parameters such as the size of the block are parsed. After all of the common parameters are parsed, method-specific parameters are loaded from the configuration file. If all of the parsings went smoothly, the parsed configuration in the form of the instance of the `SCTParams` class is returned.

The compress method from the `SCTUtils` class is called with the instance of the `SCTParams` class that holds the whole configuration for the run. This class method is responsible for creating an output file and compression chain, execution of the compression chain and for performance measurement of the execution.

The decompress method from the `SCTUtils` class firstly parses the header from the compressed file. This is necessary to gather information such as the size of the block or what chain configuration was used during the compression of the file that is supposed to be decompressed. Then the output file and decompression chain are created. The decompression chain is then executed, and its performance is measured.

The UML class diagram for the `SCTUtils` class is shown in Figure 3.9.

Figure 3.9: UML class diagram for the `SCTUtils`.



3.3 Unified file format

Up to now, every new compression method added to the library used its own file format to store data processed by it. The only way to find out which compression method was used to process the file was to take a look at the file extension and, based on that, call appropriate decompression algorithm. At the beginning of each compressed file, each method used its own header to store crucial information needed for the decompression. This approach is unsustainable for further use.

The newly implemented modular framework uses the same file format for all of the processed files regardless of the used compression methods. The `SCTParams class`, described in Subsection 3.2.2, aggregates all of the necessary methods' configurations that were used during the compression and that are essential to know for the proper decompression to happen.

Before the first write of the compressed data to the prepared output file happens, the instance of the `SCTParams` class is serialized into the JSON file format, and it is written at the begging of the output file as a header.

JSON is a well-known lightweight data-interchange file format used to transmit and save data. It is easy for humans to read and write and for machines to parse and generate, thus being a perfect candidate to serialize configuration into the header of the processed files [36].

To *serialize* an object means to convert its state to a byte stream so that the byte stream can be reverted back into a copy of the object. A Java object is *serializable* if its class or any of its superclasses implements either the `java.io.Serializable` interface or its subinterface, `java.io.Externalizable`. *Deserialization* is the process of converting the serialized form of an object back into a copy of the object [37]. The Gson library was chosen for this task. It is a Java library that can be used to convert Java Objects into their JSON representation and to convert a JSON string to an equivalent Java object [38].

The `SCTParams` class can be easily serialized and stored as a JSON string, and the `FileIO` class does this job. This approach ensures that no header processing changes will be needed when future extensions to the SCT library are added as long as the `SCTParams` class stays serializable.

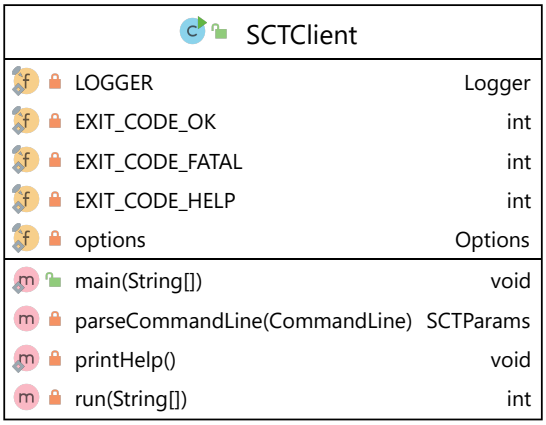
3.4 Client

Important and much-needed functionality of the SCT library is the ability to be run compression and decompression from the command-line interface. Class `SCTClient` implements this functionality. It is an executable program that can be run from the command-line interface, and its behaviour can be altered via parameters. Only common parameters, such as the size of the

block, can be configured via parameters. Method-specific parameters can only be configured via the run configuration file.

The UML class diagram for the `SCTClient` class is shown in Figure 3.10.

Figure 3.10: UML class diagram for the `SCTClient`.



The `SCTClient` class is built by running the `mvn clean install` command. It can then be run with the `java -jar sct-RELEASE.jar` command. The following text summarizes the usage of the implemented `SCTClient` class. This help message is printed if any error occurs while parsing parameters from the command line or can be purposely printed with the `-h` parameter.

```
usage: java -jar sct-RELEASE.jar input [options]
input - input file
 -b,--blocksize <block>     size of blocks in bytes -
                            default size is 1 MB
 -c,--config <config>       configuration file (if set,
                            other options are ignored and
                            config file is preferred)
 -cg,--confgen              generate default configuration
                            file to the given path
 -ch,--chain <chain>        compression chain in following
                            format: XXX-XXX-XXX-... XXX is
                            a abbreviation of supported
                            algorithm - supported algorithms
                            are: [ACB, ARI, BWT, DCA, HUF,
                            LZ77, LZ78, LZAP, LZFSE, LZMW,
                            LZW, LZY, MTF, RLE, TAAC, TBAC,
                            TFSE, TITB, ZSTD]
 -de,--decompress           decompress input (default is to
                            compress)
 -h,--help                  print help
```

```
-l,--loglevel <loglevel>   sets logging level of
                           the application
-m,--measure               measure time and compression
                           ratio and print to standard
                           output
-v,--validate              decompress compressed file and
                           compare in to the original input
```

3.5 Testing

A vital part of every software development project is testing. A proper testing framework can discover and prevent many bugs that can occur during the software development cycle. Up to now, the SCT library did not contain appropriate tests that would cover the most common but also the bordering cases that could occur during the file processing. This often led to undiscovered bugs that were indirectly caused by changes to the widely used utilities in the library, such as the `FileIO` class. Some of these problems were discovered during the work on this thesis, and they were discussed in Section 2.7.

The newly implemented testing framework for the SCT library contains 69 unit and integration tests. As of today, 2 of these 69 tests are failing. Those two are tests for the DCA method and for the LZW method. These two methods were already mentioned in Section 2.7 of this thesis as the known problems, and it was not possible to fix them in a reasonable time. All of the other known issues discussed in Section 2.7 were fixed, and the newly implemented testing framework shall prevent them from occurring again.

However, the two failing tests introduce a new problem for the compilation of the library. As it stands today, using the `mvn clean install` command to build the library's code will automatically stop the build if any test failure occurs. The best temporary solution for this problem is to run the `mvn clean install -Dtest=!DCATest,!LZWTest` command that excludes known failing tests that would otherwise block the build process.

3.6 Measurement tool

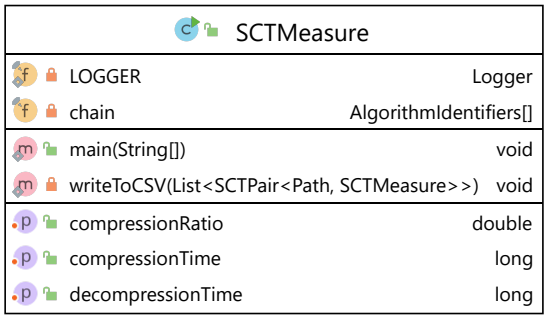
The idea behind the implementation of the measurement tool is to create a handy tool that can be used to compare the different compression methods on the same provided input. It shall accept either one input file or a whole folder that contains multiple files for processing, which is convenient when benchmarking implemented algorithms on different files from corpora.

The newly implemented benchmarking tool measures compression ratio, compression time and decompression time. These performance testing results

are saved in the CSV file format - a well-known data format that other tools can easily process later.

Class `SCTMeasure` implements this measuring tool, and its UML class diagram is shown in Figure 3.11.

Figure 3.11: UML class diagram for the `SCTMeasure`.



Chapter 4

Measurements and results

4.1 Overview

This chapter is dedicated to the performance testing of the already created implementations. The predefined processing chains, implemented in previous theses and discussed in Subsection 3.2.1, are tested on multiple corpora and time, and the compression ratio is measured. The average time and average compression ratio for all of the files from the corpus are measured. For the purpose of evaluating the capabilities of the implementations, the default block size of 1 megabyte is used. Basic overviews of the used corpora are defined in dedicated sections, followed by performance testing results. Since many other processes can affect the platform's performance, the measurements were performed multiple times, and a result with the lowest time consumed was used as a result of the measurement.

All measurements are done on the Windows 10 platform with the following configuration:

- CPU - Intel Core i7 8750H 2,2 GHz hexa-core,
- RAM - 32 GB DDR4.

4.2 Calgary corpus

The Calgary corpus is a collection of text and binary data files, commonly used for comparing data compression algorithms. It was created by Ian Witten, Tim Bell and John Cleary from the University of Calgary in 1987 and was commonly used in the 1990s. In 1997 it was replaced by the Canterbury corpus, based on concerns about how representative the Calgary corpus was, but the Calgary corpus still exists for comparison and is still useful for its originally intended purpose. It contains 18 files of 9 different types with complete size 3,266,560 bytes [17, 39].

4.3 Canterbury corpus

The Canterbury Corpus was published by Ross Arnold and Tim Bell in 1997. The aim was to replace outdated Calgary Corpus and to provide more relevant testing for new compression algorithms. The files were selected based on their ability to provide representative performance results. In its most commonly used form, the corpus consists of 11 files, selected as ”average” documents from 11 classes of documents, totalling 2,826,240 bytes [12, 17].

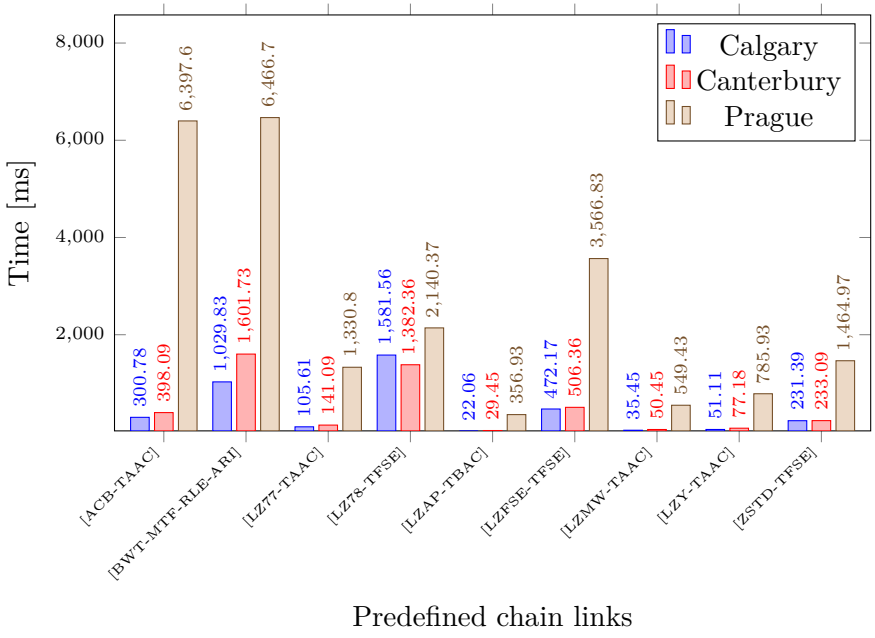
4.4 Prague corpus

The Prague corpus is specific because of its diversity. In order to keep the corpus up to date, a methodology for regular updates of the corpus was designed. Being the largest of those three corpora, this corpus contains 30 files of a total size of 58,265,600 bytes. The main intention for the creation of the Prague corpus was to create a better benchmarking tool and set the bar higher for the compression algorithms [17, 40].

4.5 Average compression time

Figure 4.1 shows how different predefined chains compare in average compression time per file for all of the files from each of the corpora.

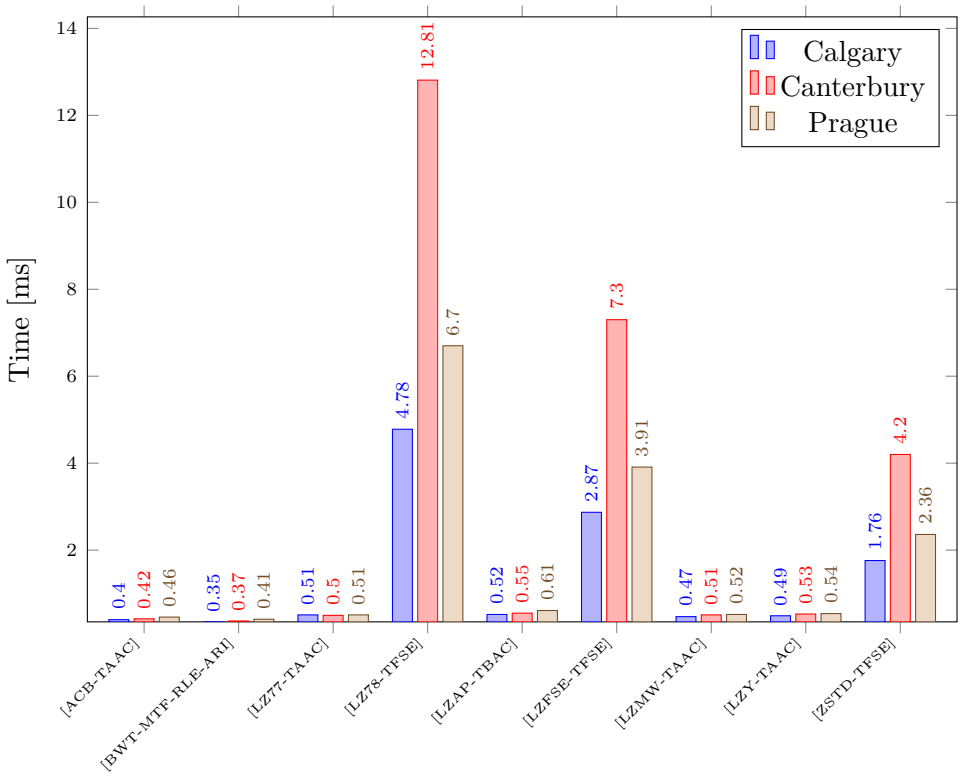
Figure 4.1: Average compression time per file for all of the files from the Calgary, Canterbury and Prague corpora.



4.6 Average compression ratio

Figure 4.2 shows how different predefined chains compare in average compression ratio per file for all of the files from each of the corpora.

Figure 4.2: Average compression ratio per file for all of the files from the Calgary, Canterbury and Prague corpora.



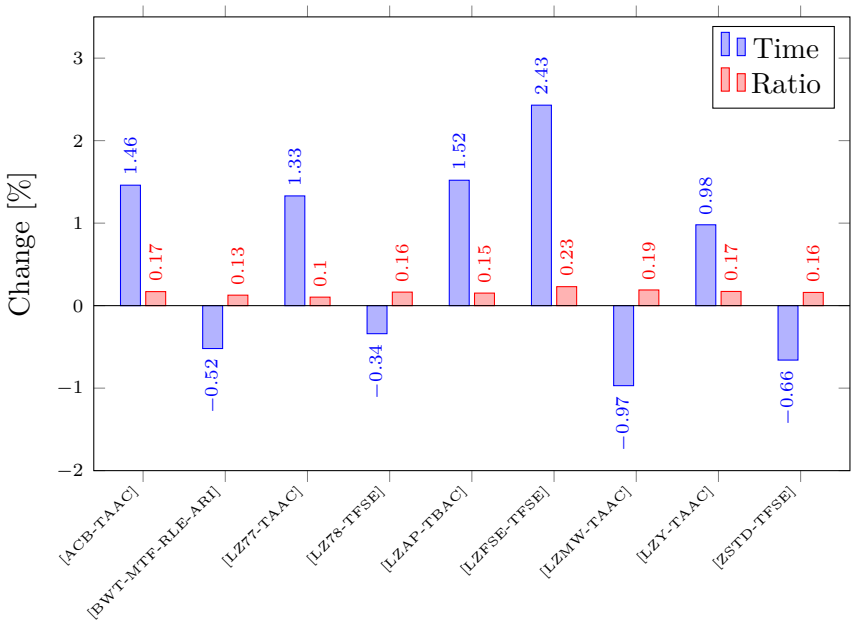
4.7 Summary

From Figure 4.1, it can clearly be observed that the main intention for the creation of the Prague corpus was successfully fulfilled and that the compression algorithms struggle to keep up the compression speed on the given data and often end up with several times worse performance than they achieved on Calgary and Canterbury corpora.

Another interesting observation emerges from Figure 4.2. Processing chains that utilise the LZFSE triplet processor apparently lag behind those that do not. An interesting topic for discussion would be to locate the source of this issue.

Another apparent topic for discussion is the change in the compression method's performance after the implementation of the new modular framework. Figure 4.3 shows how compression ratio and compression time changed compared to the old implementations available in the SCT library - an average of all files from all corpora, best result from few measurements.

Figure 4.3: Average changes in compression time and compression ratio between the old implementation and newly implemented modular framework.



Predefined chain links

A small and insignificant change in the compression ratio was expected due to the new header added to the beginning of each file. This header has a length of around 1 kilobyte (length can vary based on the length of the parameters' values in the header). The processed files from the provided corpora or in real-life situations usually have sizes many times larger, and therefore, the added header has a negligible effect on the final compression ratio. Usually, the header at the beginning of a compressed file takes up several hundredths of a percent of the total file size. However, this percentage can, for extremely small files, often present in the corpora, go up to one-quarter of the total file size, thus giving us higher average compression ratios, as are presented in Figure 4.3.

Compression time was also expected to worsen a little bit in general due to additional overhead added to the library. Few hardcoded lines for each compression method in previous implementations can easily be faster than the newly implemented and complex modular framework. However, it is still a small performance tradeoff for the robustness and clarity that it introduces

to the library.

Surprisingly, few measurements show that better results can be achieved with the modular framework, even though it introduces additional overhead to the file processing. This is attributed to the file caching in the computer memory and the fact that many other processes ran on the testing machine. The performance testing would have to be performed hundreds or even thousands of times for the more bulletproof results, instead of just a few times like it was performed now. However, the current results are sufficient for the need of this thesis and demonstrate that the performance of the implemented modules in the SCT library does not significantly worsen after the addition of the new modular framework.

Conclusion

This thesis aimed to analyze the Small Compression Toolkit library, design a new modules' interface and implement a new modular framework together with a new file format to store processed data. Maximum effort was given to fulfil this goal, and the SCT library now possesses the new modular framework and other mentioned functionalities. A completely new testing framework was added, which shall, hopefully, prevent a repetition of the same errors previously made during the library's software development cycle, such as the fact that the new changes to the library's common utilities cause malfunction previously correctly working modules.

The product of this thesis fulfils all the functional requirements and satisfy all non-functional requirements, with particular emphasis on adequately documenting the source code. All relevant elements were implemented and underwent technical measurements aimed at compression efficiency and time. The implemented functionality can be used mainly for study purposes, aiding the addition of the new extensions, but with a bit of optimization, it can also be used in real-life scenarios.

Some imperfections still exist in the current implementation. As was mentioned in the Analysis chapter, some of the implemented modules does not work correctly. It is believed that these problems were mainly caused by a nonexisting testing framework in the library when implementors of the new extensions to the library did not perform regression tests to find out if their changes to the code do not break something important used by other modules. The fact that the project does not use git's tools, such as branching, certainly does not help the case. Even though it was not part of the assignment of this thesis, a reasonable effort was made to fix malfunctioning modules in the library. Nevertheless, the DCA and the LZW modules still remain inoperational.

Future work

While working on this thesis, multiple new ideas for further improvements emerged. One of them would be parallel data processing. The newly implemented modular framework was designed with potential parallel processing in mind. This creates a significant opportunity for finally introducing parallelism into the SCT library.

Another possible future extension to the library would be a graphical user interface (GUI for short). The current implementation of the modular framework enables quite simple and straightforward implementation of GUI since it would be possible just to gather configuration specified by the user in the GUI and then call a particular method that would create and execute the defined processing chain.

The last and quite obvious possible future improvement would be to fix all of the previously implemented methods and try to keep them that way.

Bibliography

[1] Small Compression Toolkit. 2016. Available from: https://gitlab.fit.cvut.cz/polacrad/sct

[2] Data representation - Extended ASCII. 2021, [Cited 2021-11-13]. Available from: https://www.bbc.co.uk/bitesize/guides/zscvxfr/revision/4

[3] Nelson, M. *The Data Compression Book*. Wiley, second edition, 1995, ISBN 1558514341.

[4] Code. 2021, [Cited 2021-11-13]. Available from: https://en.wikipedia.org/wiki/Code

[5] Bican, J. *Implementation of the ACB compression method improvements in the Java language*. Master's thesis, Czech Technical University in Prague, Faculty of Information Technology, 2017.

[6] Data compression. 2021, [Cited 2021-11-13]. Available from: https://en.wikipedia.org/wiki/Data_compression

[7] Jain, A.; Patel, R. An Efficient Compression Algorithm (ECA) for Text Data. In *2009 International Conference on Signal Processing Systems*, 2009, pp. 762–765, doi:10.1109/ICSPS.2009.96.

[8] Data Compression Applets Library. 2021, [Cited 2021-11-13]. Available from: http://stringology.org/DataCompression/index_en.html

[9] Salomon, D. *A Concise Introduction to Data Compression*. Springer London, illustrated edition, 2008, ISBN 1848000715.

[10] Fossum, V. Entropy, Compression, and Information Content. *Information Sciences Institute*, 2013.

[11] Dollors. What is Information? Part 2a - Information Theory. *Cracking the Nutshell*, 2013, [Cited 2021-11-13]. Available from: https://crackingthenutshell.org/what-is-information-part-2a-information-theory/

[12] Powel, M. The Canterbury corpus. 2001, [Cited 2021-11-13]. Available from: https://corpus.canterbury.ac.nz/index.html

[13] IntelliJ IDEA. [Cited 2021-11-17]. Available from: https://www.jetbrains.com/idea/

[14] Novák, J. *Implementace kompresní metody DCA v jazyce Java.* Master's thesis, Czech Technical University in Prague, Faculty of Information Technology, 2018.

[15] Zemek, L. *Implementace kompresních metod LZ77, LZ78, LZW v jazyce Java.* Master's thesis, Czech Technical University in Prague, Faculty of Information Technology, 2018.

[16] Bobot, J. *Implementace kompresních metod LZY, LZMW a LZAP v jazyce Java.* Master's thesis, Czech Technical University in Prague, Faculty of Information Technology, 2019.

[17] Geletka, F. *Implementation of BWC compression method and its variants in Java programming language.* Master's thesis, Czech Technical University in Prague, Faculty of Information Technology, 2019.

[18] Zemek, L. *Finite State Entropy Coder for SCT library.* Master's thesis, Czech Technical University in Prague, Faculty of Information Technology, 2021.

[19] What is GitLab? [Cited 2021-11-17]. Available from: https://about.gitlab.com/what-is-gitlab/

[20] Apache Maven Project. [Cited 2021-11-17]. Available from: https://maven.apache.org/

[21] Apache Commons. [Cited 2021-11-17]. Available from: https://commons.apache.org/proper/commons-cli/

[22] Apache Log4j 2. [Cited 2021-11-17]. Available from: http://logging.apache.org/log4j/2.x/

[23] JUnit 5. [Cited 2021-11-17]. Available from: https://junit.org/junit5/

[24] Interface Consumer. [Cited 2021-11-20]. Available from: https://docs.oracle.com/javase/8/docs/api/java/util/function/Consumer.html

[25] Burrows, M.; Wheeler, D. A Block-sorting Lossless Data Compression Algorithm. *Systems Research Center*, 1994. Available from: https://www.hpl.hp.com/techreports/Compaq-DEC/SRC-RR-124.pdf

[26] Zalik, B.; Lukac, N. Chain code lossless compression using move-to-front transform and adaptive run-length encoding. *Sig. Proc.: Image Comm.*, volume 29, 2014: pp. 96–106.

[27] Liu, Y.; Goutte, R. Lossy and lossless spectral image compression using quaternion Fourier, Burrows-Wheeler and Move- to-Front transforms. *2012 IEEE 11th International Conference on Signal Processing*, volume 1, 2012: pp. 619–622.

[28] RLE compression. 2009, [Cited 2021-11-17]. Available from: https://www.prepressure.com/library/compression-algorithm/rle

[29] Murray, J. D.; vanRyper, W. *Encyclopedia of Graphics File Formats (2Nd Ed.)*. Sebastopol, CA, USA: O'Reilly & Associates, Inc., 1996, ISBN 1-56592-161-5.

[30] Nayuki. Reference arithmetic coding. *Project Nayuki*, 2018, [Cited 2021-11-17]. Available from: https://www.nayuki.io/page/reference-arithmetic-coding

[31] Nayuki. Reference Huffman coding. *Project Nayuki*, 2018, [Cited 2021-11-17]. Available from: https://www.nayuki.io/page/reference-huffman-coding

[32] Witten, I. H.; Neal, R. M.; et al. Arithmetic Coding for Data Compression. *Commun. ACM*, volume 30, no. 6, June 1987: pp. 520–540, ISSN 0001-0782, doi:10.1145/214762.214771. Available from: http://doi.acm.org/10.1145/214762.214771

[33] Lambert, S. *Implementing Associative Coder of Buyanovsky (ACB) Data Compression*. Montana State University–Bozeman, 1999. Available from: https://books.google.cz/books?id=K7LPNwAACAAJ

[34] Lempel–Ziv–Welch. 2021, [Cited 2021-11-25]. Available from: https://en.wikipedia.org/wiki/Lempel\OT1\textendashZiv\OT1\textendashWelch

[35] Asymmetric numeral systems. 2021, [Cited 2021-11-28]. Available from: https://en.wikipedia.org/wiki/Asymmetric_numeral_systems

[36] Introducing JSON. [Cited 2021-12-09]. Available from: https://www.json.org/json-en.html

[37] Serializable Objects. [Cited 2021-12-09]. Available from: https://docs.oracle.com/javase/tutorial/jndi/objects/serial.html

[38] Gson. [Cited 2021-12-09]. Available from: https://github.com/google/gson

[39] Calgary corpus. 2021, [Cited 2021-12-13]. Available from: https://en.wikipedia.org/wiki/Data_compression

[40] Holub, J.; Reznicek, J.; et al. Lossless Data Compression Testbed: ExCom and Prague Corpus. In *Lossless Data Compression Testbed: ExCom and Prague Corpus*, 03 2011, p. 457, doi:10.1109/DCC.2011.61.

Appendix A

Acronyms

AC Arithmetic coder

ACB Associative Coder of Buyanovsky

ANS Asymmetric numeral systems

API Application programming interface

ARI Arithmetic coder

ASCII American Standard Code for Information Interchange

BWC Burrows-Wheeler compression

BWT Burrows-Wheeler transform

CLI Command-line interface

CPU Central processing unit

CSV Comma-separated values

DCA Data Compression using Antidictionaries

FSE Finite State Entropy

GIF Graphics Interchange Format

GUI Graphical user interface

HC Huffman coder

HUF Huffman coder

IDE Integrated development environment

JSON JavaScript Object Notation

JVM Java virtual machine

LZ77 Lempel-Ziv 1977

LZ78 Lempel-Ziv 1978

LZAP Lempel-Ziv All prefixes

LZFSE Lempel-Ziv Finite State Entropy

LZMW Lempel-Ziv-Miller-Wegman

LZW Lempel-Ziv-Welch

LZY Lempel-Ziv-Yabba

MTF Move-to-front transform

OS Operating system

PDF Portable Document Format

RAM Random-access memory

RLE Run-length en-coding

SCT Small Compression Toolkit

UML Unified Modeling Language

TAAC Triplet Adaptive Arithmetic coding

TBAC Triplet Bit Array Composing

TFSE Triplet Finite State Entropy coding

TITB Triplet Integer To Bits coding

VCS Version control system

ZSTD Zstandard

Appendix B

Contents of enclosed CD

- `readme.txt` ..... the file with CD contents description
- `src` ..... the directory of source codes
- `text` ..... the thesis text directory
  - `DP_Geletka_Filip_2021.pdf` ..... the thesis text in PDF format
  - `src` ..... the directory of LaTeX source codes of the thesis