UKRAINIAN CATHOLIC UNIVERSITY

BACHELOR THESIS

EDIFACT parsing and converting library in Python

Author:
Ihor RAMSKYI

Supervisor:
Yurii PRYYMA

A thesis submitted in fulfillment of the requirements for the degree of Bachelor of Science

in the

Department of Computer Sciences and Information Technologies
Faculty of Applied Sciences

Lviv 2023

Declaration of Authorship

I, Ihor RAMSKYI, declare that this thesis titled, "EDIFACT parsing and converting library in Python" and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed:

Date:

"Done is better than perfect."

Sheryl Sandberg

UKRAINIAN CATHOLIC UNIVERSITY

Faculty of Applied Sciences

Bachelor of Science

EDIFACT parsing and converting library in Python

by Ihor RAMSKYI

Abstract

This thesis centers on the exploration of Electronic Data Interchange (EDI) data formats, with an emphasis on EDIFACT. Recognized for its complexity and arduousness in utilization, the objective of this research is to construct an efficient and user-friendly Python library for the facilitation of EDIFACT file manipulation. This library is designed with capabilities to parse, validate, and subsequently convert EDIFACT files into contemporary data formats such as Extensible Markup Language (XML) and JavaScript Object Notation (JSON). Furthermore, this thesis provides a comprehensive review of comparable open-source tools currently available.

Acknowledgements

I have a word of gratitude for my supervisor, Yurii Pryima, for helping me get through this work, as well as for my fellow students for fun years spent together

Contents

List of Figures

List of Abbreviations

ANSI	(The) **A**merican **N**ational **S**tandards **I**nstitute
CSV	**C**omma **S**eparated **V**alues
DFDL	**D**ata **F**ormat **D**escription **L**anguage
EDI	**E**lectronic **D**ata **I**nterchange
EDIFACT	**E**lectronic **D**ata **I**nterchange (for) **A**dministration, **C**ommerce (and) **T**ransport
ETL	**E**xtract, **T**ransform, **L**oad
FTP	**F**ile **T**ransfer **P**rotocol
JSON	**J**ava**S**cript **O**bject **N**otation
POJO	**P**lain **O**ld **J**ava **O**bject
TDCC	**T**ransportation **D**ata **C**oordinating **C**ommittee
UML	**U**nified **M**odelling **L**anguage
UNECE	**U**nited **N**ations **E**conomic **C**ommision (for) **E**urope
XML	e**X**tensible **M**arkup **L**anguage

Dedicated to my family, who supported me through the whole study and life

Chapter 1

Introduction

1.1 Motivation

Electronic Data Interchange (EDI) is a standard format for exchanging business documents electronically. It has become increasingly important in modern business due to its ability to reduce costs, improve efficiency, and streamline processes. EDIFACT is a widely used EDI standard that many organizations worldwide use (*The Future of EDI: An IBM point of view* n.d.).

However, EDIFACT documents can be challenging to parse and convert, especially for those unfamiliar with the format. This can make it difficult for businesses to implement EDI effectively and efficiently. As such, there is a need for tools and libraries that can help simplify the process of working with EDIFACT documents. During my work as a Python developer at a software company, I undertook the responsibility of implementing EDIFACT document parsing in Python and embedding the piece into an ERP system. Therefore, I started researching to find the proper tools to successfully execute this task and found none of good enough quality.

1.2 Goal

This bachelor's thesis aims to develop and present a Python library that can effectively parse and convert EDIFACT and EDIFACT-similar documents. We aim to create a library that is easy to use and versatile, with features such as validation and error handling often absent in open-source tools.

To achieve this goal, we will provide an overview of EDI and the EDIFACT standard and conduct an analysis of existing EDIFACT libraries to identify their strengths and limitations. Based on this analysis, I will design and implement a new library that addresses the identified limitations and provides additional features.

The library should be designed to be modular and extendable, allowing users to customize it to fit their specific needs. In addition, it will be thoroughly tested to ensure its reliability and accuracy in parsing and converting EDIFACT documents.

Throughout the thesis, we will describe the library's architecture, design, and implementation. We will also provide examples of how the library can be used to parse and convert EDIFACT documents.

Overall, the goal is to contribute a comprehensive and user-friendly Python library that can ease EDI implementation and streamline the processing of EDIFACT documents for businesses and developers.

Chapter 2

Background

2.1 EDI history

While the roots of EDI can be traced back to the 1960s, it started vaguely approaching its current state in the 1970s (Zavorskas, 2020). In 1968, TDCC was organized to develop standard formats for exchanging business information. In 1975, the first EDI specifications, Rail Transportation Industry Application, was released, which allowed reducing the amount of paperwork in the railroad industry. It was also supported by FTP, which was introduced in 1971.

The first try was a success, and it led to working on new standards for other industries, so ANSI chartered ASC (Accredited Standard Committee) X12, and in 1981 they issued standard ANSI X12, which is currently traditional for North America. It did not fully satisfy the need of the European business, so in 1985 United Nations released standard UN/EDIFACT to solve the problem (Waisman, n.d.), which is currently traditional for Europe and Asia.

Later, more modern data formats like XML(1996) and JSON(2001) were developed and started gaining popularity. They are easier to read, parse and validate, but ANSI X12 and EDIFACT are still actively used and updated.

2.2 General information about EDIFACT

EDIFACT is an international standard for EDI, most common in Europe and Asia. It is widely used in various industries, such as transportation, logistics, finance, and trade, to ease the exchange of structured business documents between trading partners. It was designed to get rid of paperwork and delegate as much as possible of document management to computers (Waisman, n.d.).

EDIFACT provides a standardized format for representing business data in a machine-readable form. It defines a set of rules and syntax for creating structured messages that can be exchanged between computer systems.

Each EDIFACT message has its type by the purpose and structure of the message. The most common message types are orders, invoices, shipping instructions, and payment instructions. Each message type has a predefined structure, explicitly described in the documentation but implicit in the message itself. (*UNECE* n.d.)

EDIFACT messages are usually transmitted using different communication protocols, including value-added networks (VANs), FTP or its derivatives, AS2, or web services (*The Four Most Common EDI Protocols* n.d.). The receiving system has to interpret the EDIFACT message and extract the relevant data for further processing, such as updating inventory, generating invoices, or triggering fulfillment processes (*KITS EDI technical documentation* n.d.).

2.3 EDIFACT document structure

EDIFACT document structure is hierarchical. There are certain units of information (*UNECE* n.d.):

1. Element
2. Composite
3. Segment
4. Segment Group
5. Message
6. Functional Group
7. Interchange

Which include each other according to the diagram below (the data unit it points to):

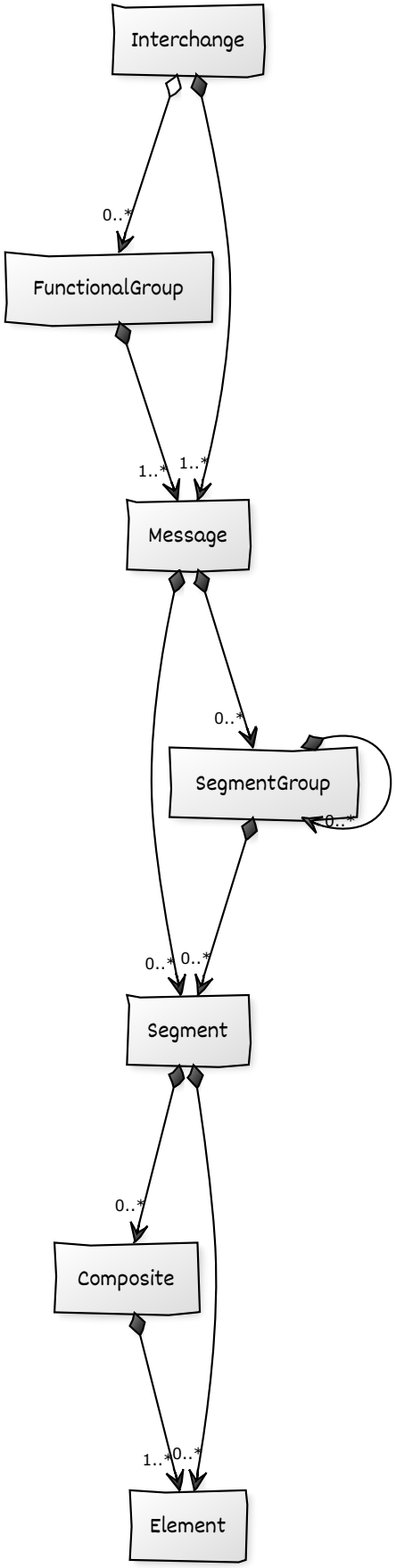

FIGURE 2.1: EDIFACT document structure

2.4 File examples

Here is an EDIFACT document example:

```
UNB+UNOA:2+MYCHAINCM:ZZ+925485MX00:8+080909:1357+807'
UNH+000012102+INVOIC:D:96A:UN:EAN008'
BGM+380+ABCD1234+9'
DTM+137:19980110:102'
RFF+ON:0123456789123'
NAD+BY+1111111::31B'
NAD+PE+2002450:31B'
NAD+SU+3333333:31B'
LIN+1'
PIA+5+0835201031:IB'
QTY+47:2'
PRI+AAA:8.4'
PRI+AAB:10.5::SRP'
ALC+A'
PCD+3:20'
LIN+2'
PIA+5+0835208338:IB'
QTY+47:1'
PRI+AAA:24'
PRI+AAB:30::SRP'
ALC+A'
PCD+3:20'
UNS+S'
MOA+86:40.8'
CNT+2:2'
CNT+1:3'
UNT+26+000012102'
UNZ+1+807'
```

Here data units are:

- Segment: each line - from three uppercase letters to segment terminator apostrophe (') -inclusively. It can consist of composites and elements separated by plus (+)
- Composite: consists of elements (can have only one), elements inside separated with a colon (:). If a composite consists of only one element, it has the same notation as if it would be an element - without only colons.
- Element - the smallest data unit, part of segment, or composite.
- Segment Group - consists of segments and segment groups of lower order. Described only in specifications and has no special notation in the EDI file.
- Message: all segments from UNH to UNT inclusively
- Interchange: the whole document

That way, `DTM+137:19980110:102'` is a segment, `137:19980110:102` is a composite, and each of `137`, `19980110`, and `102` are elements.

As you can see, the structure is quite inconsistent, complex, and difficult for a human to read. Also, the Segment Groups cannot be seen without knowing the structure of the EDIFACT Message beforehand, resulting in an inability to validate the message without storing the structure somewhere else.

2.5 EDI vs XML

That was one of the reasons why modern standards like XML currently grow at a pace not slower than EDI. Even though EDI has longer history and tradition, and there are already a lot of companies that use it, XML is gaining popularity, especially in smaller, newer companies, for having easier semantics and more instruments to work with. However, they are both used a lot and have their place in business. (Nurmilaakso, 2008)

Chapter 3

Approach to solution

3.1 Problems

To create a useful library, it should solve problems that user typically encounters working with EDI/EDIFACT. After researching and from our own experience, we can formulate those problems:

1. Files are not designed to be human-readable
2. Files are hard to work with and have implicit structure
3. Validating the file is not a trivial task

3.2 Related works

3.2.1 Python works

The pool of open-source Python tools for working with EDI/EDIFACT is relatively poor. These are the most popular ones:

- `https://github.com/notpeter/edicat` is a small program, installable via pip, that only adds newlines between segments. Does not solve any of the above-mentioned problems.
- `https://github.com/bots-edi/bots` is a program that supports multiple EDI standards, including EDIFACT, parses and validates them. However, it is not a library, so it cannot be used for specific needs and to be included well enough in another program.
- `https://github.com/FriedrichK/python-edifact` is a toolkit for working with EDIFACT files. It provides basic validation, but it is unfinished, unmaintained (no updates for 7 years now as of 2023), lacks documentation, and is for Python2 only.
- `https://github.com/nerdocs/pydifact` is a lightweight library for EDIFACT basic (unfull) parsing and validation.

3.2.2 Non-Python works

However, there are good implementations of the tool in other languages. The most notable are:

- https://www.smooks.org is a huge framework in Java for working with a vast amount of kinds of text formats - EDI, XML, CSV, POJO, DFDL, and even Java classes, especially with huge files. It uses streaming data to process text in chunks, not all at once. The framework works with so-called "plugs" and "cartridges," where plugs implement different kinds of processing data and cartridges set the rules by which plugs work. It makes Smooks flexible with its modular and extensible architecture. However, it is quite heavy-weight.
- https://github.com/jf-tech/omniparser/ is an ETL parser in Go as well as Smooks and uses a similar approach. It uses user-generated schemas to transform files according to them. However, it cannot be used as a library and converts everything only to JSON.

Upon going through these instruments, it became clear that not a single one of them suits my needs fully, and there is a need to create a good one.

3.3 Solution approach

Meanwhile, the JSON representation of the same document will be much easier to read for a human, even though the text appears much longer. Example: single Segment from the Interchange example in Chapter 2:

```
NAD+BY+1111111::31B'
```

in JSON representation look this way (it is not enclosed because the representation of a single Segment is taken from a full JSON file) :

```
"Name_and_address": [
    {
        "Party_qualifier_1": "BY",
        "Party_identification_details_2": {
            "Party_id__identification_1": "1111111",
            "Code_list_qualifier_2": "",
            "Code_list_responsible_agency__coded_3": "31B"
        }
    }
]
```

While qualifiers and identifications might still confuse a new EDIFACT user, the names of JSON keys give a hint of what is going on.

XML representation is quite similar:

```
<Name_and_address>
    <Party_qualifier_1>BY</Party_qualifier_1>
    <Party_identification_details_2>
        <Party_id__identification_1>1111111</Party_id__identification_1>
        <Code_list_qualifier_2/>
        <Code_list_responsible_agency__coded_3>31B
            </Code_list_responsible_agency__coded_3>
    </Party_identification_details_2>
</Name_and_address>
```

Another example:

```
PIA+5+0835201031:IB'
```

JSON:

```
"Additional_product_id": [
    {
        "Product_id__function_qualifier_1": "5",
        "Item_number_identification_2": {
            "Item_number_1": "0835201031",
            "Item_number_type__coded_2": "IB"
        }
    }
],
```

XML:

```
<Additional_product_id>
    <Product_id__function_qualifier_1>5</Product_id__function_qualifier_1>
    <Item_number_identification_2>
        <Item_number_1>0835201031</Item_number_1>
        <Item_number_type__coded_2>IB</Item_number_type__coded_2>
    </Item_number_identification_2>
</Additional_product_id>
```

Therefore, we decided to solve each problem with such steps:

- Library should include a pre-generated scheme that will be used to parse and validate the document. That way, the structure is explicitly stated and used by the library.
- Result of parsing must be convenient to use.
- Library should support converting the document into easier-to-read and more popular nowadays formats: XML and JSON.

Chapter 4

Solution

4.1 Alternative solution

4.1.1 Omniparser

The most significant source of inspiration for this work among the abovementioned was omniparser. The ideas used in it provided valuable insights for working with EDIFACT, even though it was developed in Golang. It is also the most popular open-source EDI processing tool. The main insight provided was using a scheme for parsing. Even though the idea lies on the surface, the example revealed that such an approach is possible and probably optimal. The scheme that omniparser uses for EDI relies on creating a tree of nodes with a node representing a data unit. Its format is JSON, and it is able to fully describe the structure of the document. It includes:

- Parsing information (file_declaration) - specifies the rules to process the document:
  - Settings for the whole document (Element/Composite/Segment delimiter).
  - Virtual Segment declarations (Segment Groups).
  - Segment declarations.
  - Element/Composite declarations.

  Parsing information also includes accompanying information, such as the minimal/maximal count of the same Segments/Segment Groups in a batch.
- Transform information (transform_declaration) - specifies what information must be the result of the transformation:
  - Transforming templates.
  - Constants.
  - Paths to parsed fields.
  - Types of resulting values.

We did not fully describe the full capabilities of this tool as they cover more functionality that we learned to reuse in the EDIFACT-only library. More information about it can be found by the repository link: `https://github.com/jf-tech/omniparser`.

4.1.2 Difference to existing solutions

Considering the existence of such well-done tools already, the question of why bother creating an alternative arises. The main difference in our proposed solution is the

flexibility. The omniparser is a very flexible toolkit to deal with different types of data, including EDIFACT as just one part of it. This flexibility enables a user to handle diverse data formats and structures. However, it is necessary to manually create a scheme for the described above structure for the format, even, preferably, for each new document, which can be a time-consuming task. In contrast, our solution takes a different approach. Although it s a more defined and constrained framework, it aims to deliver enhanced power and functionality for a specific task. One of its key advantages is that it will be designed to be user-friendly right from the start. By including a predefined scheme within the solution's architecture, we aim to minimize the extra effort required from users. This means that users can utilize the library without having to do tedious manual work.

Also, aside from omniparser being implemented in Golang, it is not a library. It is a program. Therefore, it is a questionable decision to use it as a part of other programs. With a need for a library in Python, the easiest way to embed omniparser into existing products would be developing a Python wrapper for it, which would create many inconvenient dependencies, require interprocess communication, and lead to bad design decisions. Henceforth, we limited the usage of the program to only get inspiration from good ideas and change them to the best of our needs.

4.2 Parsing and validation

Pydifact, the library on the base of which we developed the solution described below, is capable of doing basic parsing and validation. Validation includes checking that document meets the EDIFACT syntax rules such as:

- Segment's tag must consist of 3 uppercase English letters
- the Segment is divided into Composites and Elements by plus signs (or whatever is stated in the UNA Segment)
- Composites are divided into Elements by colon signs (or whatever is stated in the UNA Segment)
- the Segment ends with an apostrophe (or whatever is stated in the UNA Segment)

These are all possible validations without saving information about data unit restrictions implicit in the document. While this does not fully cover possible document incorrectness, it gives us the basis to continue.

The parsing alongside these validations creates an Interchange object consisting of Messages and Segments. Segments, as objects, consist of lists of strings representing Composites and strings representing Elements. Considering it is not all the necessary checks and parsing, we will call this object preparsed (and prevalidated).

The extension takes this preparsed object and parses it further instead of working with the raw document. An object that fully and correctly describes the EDIFACT document requires a more complicated scheme.

Because of their complex relationships, we created a separate class for each data unit. Each class had specific fields needed for parsing and validation, like tag, version, and parent in the tree hierarchy, and a field containing objects of direct children.

An Interchange consists of multiple messages that have no relation to each other, so they are put in a list (Functional Groups support is yet to be implemented). A Message consists of Segment Groups and Segments that can be repeated (have the

same tag). Segment Group has the same relation to data units as Message. Hence, its contents field is OrderedDict, where its key is a tag of a data unit, and its value is a list of a possibly repeatable data unit. A Segment consists of unrepeatable Composites and Elements, so its contents are a list of children data units. A Composite can have Elements repeatable, so it is again an OrderedDict with Element's tag as a key and a list of Element objects as value. Finally, Elements's content is only its string value.

To make sure that preparsed by pydifact interchange is correct, there also must be other validations:

- Each Element consists of characters of its designed type. It can be numeric, alphabetic, or alphanumeric.
- Each Element's value length is in its defined range (minimum and maximum character numbers limit it).
- Segment Groups/Segments/Composites/Elements are arranged in the correct order
- All mandatory Segments/Composites/Elements are present
- All Segments and Segment Groups repeat no more than the allowed number of times
- All these validations are only possible by using values and order from the documentation, as they are not present in EDIFACT documents explicitly. Hence, we decided to write scripts that parse the documentation files, extract data from them and store it to access while creating objects.

Validation of the content of Elements, Composites, and Segments is trivial. When creating objects of these data units, restrictions are pulled from the dict by the tag of the data unit.

Parsing and validation are tightly connected when validating the Segments' and Segment Groups' order. Each Message of Interchange is processed separately. We pull respectful Message structure from parsed data from documentation files. Since the depth of each Segment is unknown before parsing, we took a recursive approach and kept the Message structure extracted from the documentation in parallel. Parsing each Segment is done in this way:

- Remember the currently parsed Segment from the document.
- Look up the current tag of the Message structure.
- If they are equal, finish parsing the current Segment and take the next one, subtract 1 from possible repetitions of this Segment
- If they are unequal, try to take the following line in the Message structure.
- Otherwise, take the next Segment in the Message structure. If it was the last one in the current Segment Group, then if any Segment from the document was processed, return to the start of Segment Group; else, take the following line. This way, eventually, parsing achieves the end of the EDIFACT document or the Message structure. If it is both, parsing and validation ended successfully. If the document ends first, parsing is successful if no mandatory segments are left in the structure. Otherwise, the document is not suitable for the structure.

Finally, the Interchange scheme approximately looks like this:

```
[ # Interchange object
    { # Message object
        "SegmentGroup_tag": [
            { # Segment Group
                "SegmentGroup_tag": [...]
                "Segment_tag": [
                    { # Segment object
                        "Composite_tag": [
                            { # Composite object
                                "Element_tag": Element_object,
                                "Element_tag": Element_object,
                                ...
                            }
                        ],
                        "Element_tag: Element_object,
                        ...
                    },
                    {...}
                ]
            },
            {...}
        ]
    },
    {...}
]
```

All the classes that represent different EDIFACT data units are shown on the UML class diagram below:

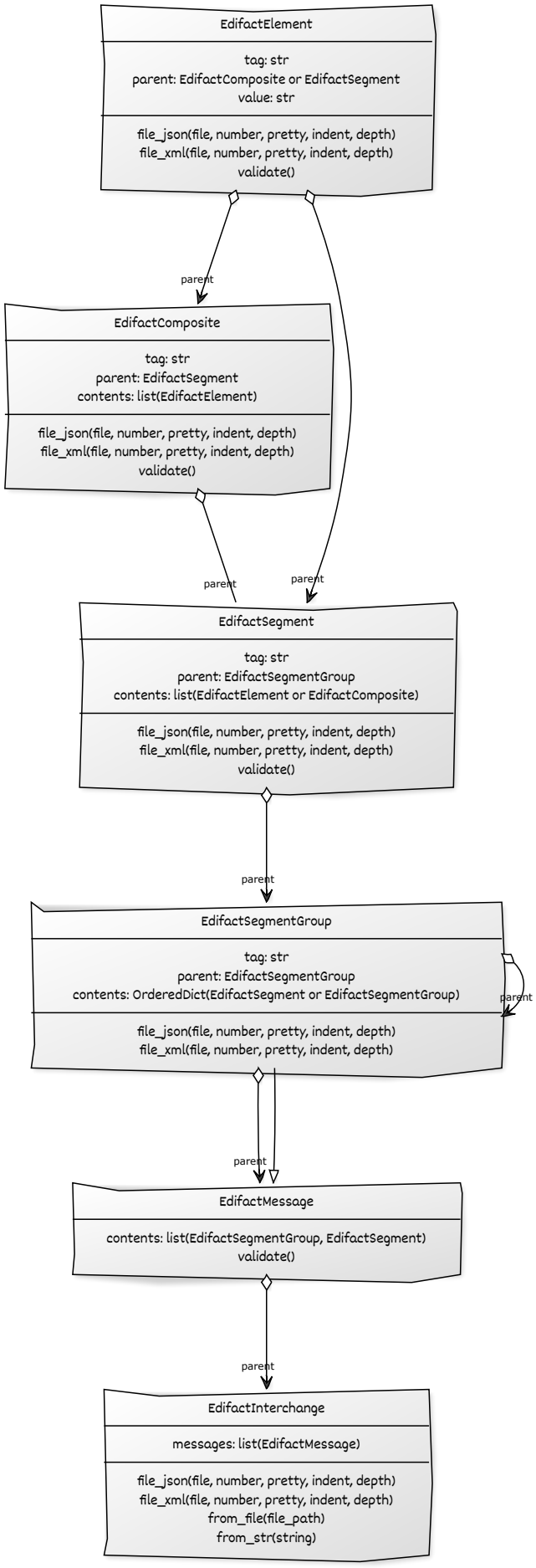

FIGURE 4.1: UML diagram of classes, representing data units and their relations. Shows their attributes and methods

4.3 Conversion

After parsing and validation at the same time, conversion to JSON and XML becomes rather trivial, considering that everything is formed. To avoid using extra memory, methods `file_json()` and `file_xml()` that we implemented write straight to the file. They are recursive methods - if called from an Interchange object, it will call `file_json()` of `file_xml()` respectively from Message and so on up to the Element.

4.4 Results and future work

Implemented library is located by the link `https://github.com/botanich/pydifact`. Work could become better with some extensive testing, adding support for conversion to CSV - because it is a popular format in business as well.

Another good feature would be adding the support of streaming data. The reason for that is that while files used tend to be small, the standard allows the creation of huge valid messages weighing tens of gigabytes.

Also, a good idea would be to add support for functional groups.

To make this library more convenient, we should also create a webpage with comprehensive documentation.

And to finish the flow of development, we should consider merging our work to the original repository of pydifact. It, though, requires help from its author.

Chapter 5

Conclusion

The goal of this work was to solve the problems with using EDIFACT - it is difficult for a human to read, has many concealed and implicit data, and is hard to validate. Our new library successfully solves them. It helps to validate and parse EDIFACT files in Python programs as well as just convert them to something more convenient to work with using such modern data formats as JSON and XML. Result of our work can be used by the link https://github.com/botanich/pydifact.

Bibliography

Daniel Furman, Jason Capriotti (2020). *bots-edi*. URL: https://github.com/bots-edi/bots (visited on 05/15/2023).

González, Christian (2023). *pydifact*. URL: https://github.com/nerdocs/pydifact (visited on 05/15/2023).

JSON: JavaScript Object Notation (2019). The JSON Data Interchange Syntax. URL: https://www.json.org (visited on 05/15/2023).

Kauder, Friedrich (2016). *python-edifact*. URL: https://github.com/FriedrichK/python-edifact (visited on 05/15/2023).

KITS EDI technical documentation (n.d.). KITS. URL: https://vendor.kingfisher.com/media/1021/kits-edifact-store-customer-po-technical-specification-130818.pdf (visited on 05/15/2023).

Nurmilaakso, Juha-Miikka (2008). "EDI, XML and e-business frameworks: A survey". In: *Computers in Industry* 59.4, pp. 370–379. ISSN: 0166-3615. DOI: https://doi.org/10.1016/j.compind.2007.09.004. URL: https://www.sciencedirect.com/science/article/pii/S0166361507001455 (visited on 05/15/2023).

Simon Liang, Steven Wang (2023). *omniparser*. URL: https://github.com/jf-tech/omniparser/ (visited on 05/15/2023).

smooks (n.d.). URL: https://www.smooks.org (visited on 05/15/2023).

The Four Most Common EDI Protocols (n.d.). URL: https://ecosio.com/en/blog/the-4-most-common-edi-protocols/ (visited on 05/15/2023).

The Future of EDI: An IBM point of view (n.d.). IBM. URL: https://www.ibm.com/watson/supply-chain/resources/future-of-edi/ (visited on 05/15/2023).

Tripp, Peter (2018). *edicat*. URL: https://github.com/notpeter/edicat (visited on 05/15/2023).

UNECE (n.d.). UNECE. URL: https://unece.org (visited on 05/15/2023).

Waisman, Abraham (n.d.). *EDI History*. URL: https://www.laits.utexas.edu/~anorman/BUS.FOR/course.mat/Flores.Project/waisman/page2.1.html (visited on 05/15/2023).

Zavorskas, William (2020). *A HISTORY OF EDI*. URL: https://www.linkedin.com/pulse/history-edi-william-zavorskas/ (visited on 05/15/2023).