Design and implementation of fully configurable interpreter and generator of HL7 Standard protocol messages

by
Robert Guziołowski

Poznan University of Technology, Poznan, Poland

Table of Contents

List of Figures

List of Tables

1 Introduction

Organization and delivery of healthcare services always contain a lot of information processing and exchange. The more of these processing and exchange is automated, the more efficent the healthcare operations are. The problem that arises is the compatibility of data exchanged. One of the standards concerning healthcare-related data exchange is the **Health Level 7 Standard**, or shortly, the **HL7 Standard**, which defines a communication protocol based on messages exchange. Nevertheless, it is not easy to use as it contains a lot of possible messages, which can appear in different versions. Thus, a framework for fast developing applications which use HL7 Standard protocol is needed.

This project concentrates on the design and implementation issues of such a framework. All data structures composing messages defined by the HL7 Standard are being studied. Proposed representation of this data allows to store all the defined messages with theirs variations, as well as gives the User the ability of adapting defined messages to protocol changes or creating completely new messages. All message processing is then based on this representation.

1.1 Project goal and scope

The aim of this project is to provide a functional, fully configurable interpreter and generator of version 2.3.1 of the HL7 Standard protocol messages. Both interpreter and generator should be adjustable to the needs of software or the User which uses them, as well as be able to parse and generate user-defined messages. Software should be well structurized in order to allow its further development for newer versions of the HL7 Standard protocol.

Developed software is not responsible for the processing of the data contained in the interpreted and/or generated messages. Specific data values, which are defined by the HL7 Standard, can be asserted with the predefined values. Nevertheless, both checking method and predefined values are user-configurable.

1.2 Thesis structure

The structure of this thesis is the following. Chapter 2 deals with background information related to current and previous versions of HL7 Standard. It also provides a more detailed description of protocol messages and structures used within them.

In Chapter 3 we describe the design of the software developed during the course of this project, with a special emphasis on configuration issues. Description of the representation of the data defined by the HL7 Standard is provided in great details.

Chapter 4 gives details about the implementation of developed software during the course of this project and the framework which it creates. Selected implementation problems are also discussed.

Finally, Chapter 5 concludes the thesis.

1.3 Typographic conventions

A number of typographic conventions are used throughout this thesis to make the reading of the text easier. Words or phrases *in italics* are particularly important for proper understanding of the surrounding context and should be paid special attention to. Italics are also used to place emphasis on a word. First occurrences of terms or definitions will be denoted by **bold face**.

Margin notes and references

Margin notes are used to mark important concepts being discussed in the paragraphs next to them. They are also to help the Reader scan the text. Reference numbers of all figures in the text are composed of the number of the chapter they appear in and the consecutive number of the figure within the chapter. Thus, the fifth figure in the fourth chapter would be referenced as Figure 4.5. Square brackets denote citations and references to other articles, books and Internet resources listed at the back of this thesis.

2 Background

This section provides background information related to the project. We start with a general description of the HL7 Standard, its aims and history of development. Then, we concentrate on the version of the Standard which was studied and used in this project, providing details of messages and its elements.

2.1 HL7 Standard origins

Need of standard The increase of the amount of data being processed and exchanged between healthcare facilities (i.e. hospitals, pharmacies, etc.) imposed development of many applications which automate some aspects of these data management. Unfortunatelly, these applications have been developed by different vendors causing data representation to be incompatible between different institutions. The problem in fact appeared when these institutions were obliged to exchange information between them. Thus, a need for a commonly accepted standard, which would unify transaction and communication data structures, emerged.

HL7 Standard Development of HL7 Standard, which would face problems described above, started in March 1987, and continued through the following years, resulting in several propositions and standards (see: [HL7 Web page]).

The primary goal of all HL7 Standards is to provide a standard for data exchange among healthcare computer applications (see: [HL7 Standard specification]). Apart from that, the widest variety of technical environments and evolutionary growth of the data structures should be supported.

The great number of application concerning the HL7 Standard implementation exists, i.e. [Chameleon] — for building the HL7-enabled applications, [Iguana] — for facilitating the integration of different HL7-enabled applications, or simple applications like [Scan7] — for parsing and viewing the data of the message in a User-friendly format.

2.2 HL7 Standard version 2.3.1

Version 2.3.1 Version 2.3.1 of the Standard addresses and documents all the mistakes and inconsistencies discovered in previous versions, i.e. 2.0, 2.1, 2.2, and 2.3. It is also the most recent version, which documentation could be found in the Internet in the [HL7 Web page] without charge. In the following sections the HL7 Standard version 2.3.1 will be called shortly the HL7 Standard.

Message and its components The HL7 Standard defines a set of **messages** used to exchange data between healthcare facilities as a string of characters. Each of the messages consists of **data fields** of variable length. Fields are separated with the use of a **field separator character**, and are allowed to repeat. Data fields are combined into logical groupings called **segments**; they start with three-character literal value and are identified by it within the message. Segments are separated with the use of **segment separator character**. Segments can be defined as optional or required,

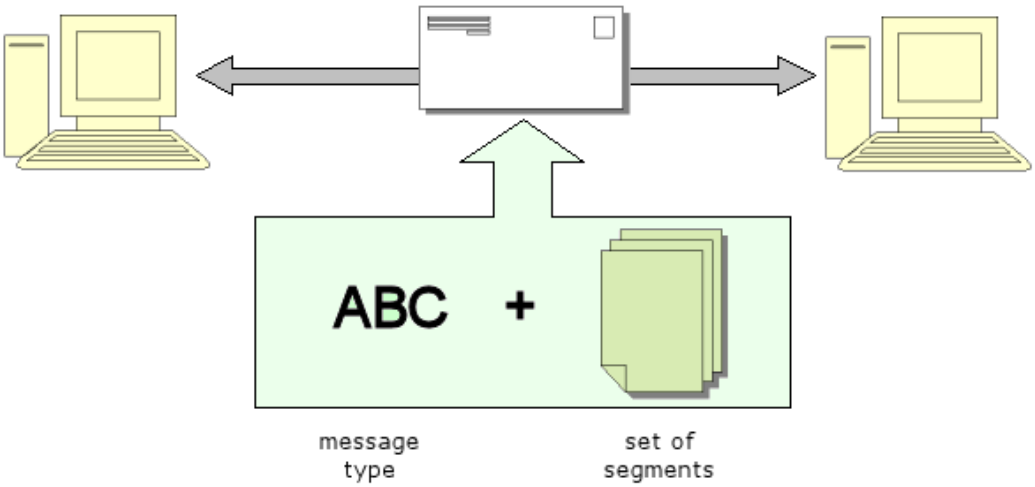
as well as allowed to repeat. Individual fields can be found in the messages by their position within associated segment. Details follows.

2.2.1 Messages

Message definition

A message is the atomic unit of data transferred between communicating systems. It is composed of a set of segments which appear in a defined sequence. Each message has a **message type** that defines its purpose. Message type is a three-character code contained within all the messages. See: Figure 2.1.

Figure 2.1 *HL7 message*



> **Warning**
>
> Figure 2.1 is a simplification; in fact, message type is contained in one of the segments of the message.

Messages' local variations

HL7 Standard reserves all messages types starting with character "Z" for locally-defined messages. The Standard supports exchange of such messages, but does not define them.

Example

ACK message type is used to transmit a "General acknowledgment message".

The full list of message types can be found in section A.2 of Appendix A of [HL7 Standard specification].

2.2.2 Segments

Segment definition

Segment is a logical grouping of fields. Segments are the only structure that can build a message directly. Segments can be defined as required or optional within the message, as well as allowed to repeat. Each of the segments is given a name — a three-character code uniquely identifying the segment. This code is know as **segment ID**. See: Figure 2.2.

Segments' local variations

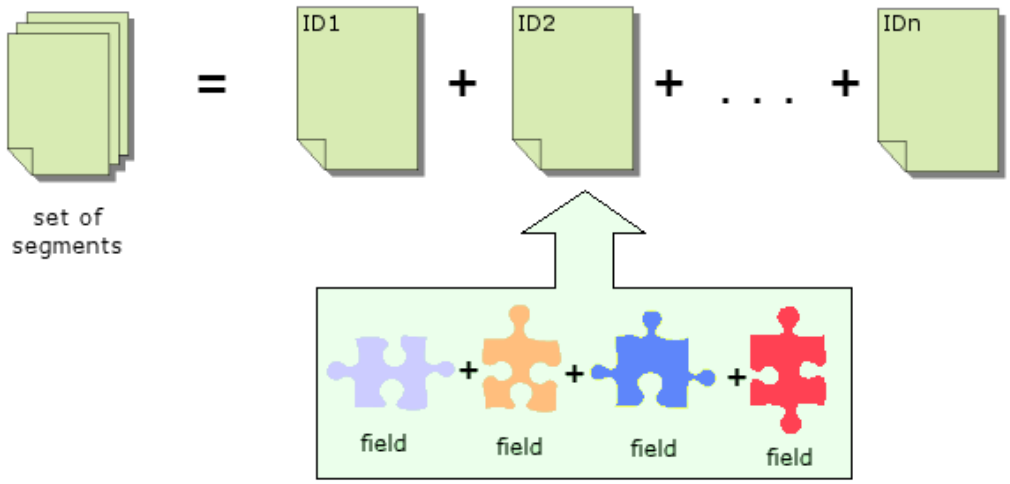
HL7 Standard reserves all segments ID starting with character "Z" for locally-defined segments. Standard supports exchange of messages containing such segments, but does not define them.

Example

Following previous example, ACK message contains following segments:

- MSH - required, cannot repeat, seqence order: 1,
- MSA - required, cannot repeat, sequence order: 2, and
- ERR - optional, cannot repeat, sequence order: 3.

Figure 2.2
HL7 segment



The full list of segments can be found in section A.3 of Appendix A of [HL7 Standard specification].

2.2.3 Fields

Field definition A field is a string of characters. Each of the fields within a message has to be a part of a specified segment. A field is allowed to repeat or to be optional. Repetition of the field can be unlimited, or specified. Within a segment, a field has its sequence number. For all properties of a field, see Table 2.1.

Predefined fields values HL7 Standards defines a set of tables containing predefined field values (see: Section 2.2.6). In such case a value contained in the field should be compared with the values appearing in the appropriate table.

Null value Fields are allowed to contain a null value: it is send as double quote marks (""), and it is different than omitting an optional field (see: Section 2.2.4 and [HL7 Standard specification]).

Coponents and subcomponents The field can be divided into components and/or subcomponents. The division is forced by the data type of the field, and is only a logical division.

Table 2.1
Field properties

Property name	Property description	Denote as
Position (sequence within the segment)	Ordinal position (sequence) of a field within a segment.	SEQ
Maximum length	Maximum number of characters which can be contained in a field. This value is normative, but can be negotiated between collaborating parties. Values provided by the HL7 Standard include counting of component and subcomponent separators (see: Section 2.2.4). Nevertheless, they are assumed for a single occurence, thus repetition separator is not included in counting of maximum length.	LEN
Data type	Restrictions on the field contents. The HL7 Standard defines 52 different data types, which are presented in more details in Section 2.2.5.	DT

Property name	Property description	Denote as
Optionality	Whether the field is required or not in the segment. The optionality of the field is defined as follows: • **R** — required, • **O** — optional, • **C** — conditional on the message type or on some other field(s), • **X** — not used in sepcified message type • **B** — left for backward compatibility with previous versions of the HL7 Standard protocol.	OPT
Repetition	Whether the field may repeat or not. The repetition of the field is defined as follows: • **N** — no repetition, • **Y** — the field can repeat an indefinite or site-determined number of times, • ***(integer)*** — the field may repeat up to the number of times specified by the integer value (denoted here as "#").	RP or RP/#
Table	Optional number of table containing predefined values; a field has to contain a value existing in specified table to be valid. If there is no table number specified for the field, it can contain any valid value.	TBL
ID number	Integer which identifies field within the HL7 Standard.	ITEM
Name	Descriptive name for the field.	ELEMENT NAME

2.2.4 Message delimiters

Separators As messages are a set of characters, certain special characters have to be reserved in order to recognize separate segments and/or fields. The HL7 Standard defines 6 special characters, wich are: **segment terminator**, **field separator**, **component separator**, **subcomponent separator**, **repetition separator**, and **escape character**. Nevertheless, all of the above **message delimiters**, except segment separator, can be defined differently for each of the messages in its message header (MSH) segment.

Message delimiters defined by the HL7 Standard are as follows:

- **Segment terminator** — Terminates a segment. This character cannot be changed and it is always a *carriage return* (<cr>, or in ASCII: hex 0D).

- **Field separator** — Separates two adjacent fields within the segment, as well as segment ID from the first data field in each of the segments. Default character: |
- **Component separator** — Separates adjacent components of the field, if exist. Default character: ^
- **Subcomponent separator** — Separates adjacent subcomponents of the field, if exist. Default character: &
- **Repetition separator** — Separates multiply occurences of the field, if exist. Default character: ~
- **Escape character** — Escape character for use in any of text fields for representing one of all 6 message delimiters. Default character: \

Omitting separators In the case when field or its component or subcomponent is optional, some of the separators may be omitted. It is possible only if after omitting such separator the field can still be parsed properly. Omitting separators is optional.

Example Field containing an address (type AD; for field types, see: Section 2.2.5) consists of 8 components: street address, other designation (second line of address), city, state or province, zip or postal code, country, address type, and other geographic designation.

Omitting unnecessary separators is presented on Figure 2.3 where the same address is represented in 3 different, but equivalent ways. Parsing all of these represetnations provides to the same result.

Figure 2.3 *Omitting separators example*

```
|Piotrowo 3a^4th floor^POZNAN^WLKP^60-965^POLAND^^|
|Piotrowo 3a^4th floor^POZNAN^WLKP^60-965^POLAND^|
|Piotrowo 3a^4th floor^POZNAN^WLKP^60-965^POLAND|
```

Nevertheless, omitting a separator causing an ambiguity between data assignments to field components is not allowed. For example, omitting a second component (*other designation*) in the example above in the way shown in Figure 2.4, point a) is incorrect, as value "POZNAN" will be recognized as *other designation* component. Proper way of omitting the component is shown in Figure 2.4, point b).

Figure 2.4 *Omitting a component example*

```
a)   |Piotrowo 3a^POZNAN^WLKP^60-965^POLAND|
b)   |Piotrowo 3a^^POZNAN^WLKP^60-965^POLAND|
```

2.2.5 Data types

Data type definition Data type of a field put restrictions on the values which a field can contain. The HL7 Standard defines 52 different data types, which include simple data types (strings, formatted strings, data, time, time stamps, or numeric) and complex data types (defined as a sequence of simple data types or other complex data types).

More details about data types definitions can be found in [HL7 Standard specification].

2.2.6 Tables

Tables definition HL7 Standards defines two sets of tables containing predefined field values. Values contained in this tables are the only values which the field with assigned table number can contain.

Nevertheless, the HL7 Standard does not define how nor when to check the values of these fields.

Standard-defined and user-defined tables

The first set of tables which the HL7 Standard defines is set of standard-defined tables. These tables are assigned a number, a name, and all the possible values. The second set is the user-defined set of tables, which are defined by the HL7 Standard as tables with assigned number and name. Values for this tables are user-defined and can differ in different institutions.

For all the HL7 Standard defined tables and their values refer to Sections A.4 and A.5 of Appendix A of [HL7 Standard specification].

3 Design

This section gives detailed information about design of the created software. At the beginning general design image of created software is presented and discussed. Next, information concerning storing software configuration data is provided. Then, following two sections give detailed configuration issues of interpreter and generator, as well as of parsing and generation methods. Finally, optional configuration concerning storing parsed data in a database, or using a database as a source of data to generate messages, is presented.

3.1 General design

HL7API library This section gives information concerning general design of created software. At the beginning it discusses the idea of creating the **HL7API library** (or shortly: HL7API) than a specific software. Afterwards, details of main modules building the HL7API library are presented.

3.1.1 HL7API library

Why library? The HL7 Standard protocol is designed to transfer data between communicating systems in the way of well-structured messages. It provides no information concerning storing the data in the system, nor processing them. Thus, building a specific software depends to a great extent on the goals which the software has to accomplish, and creating such a software limits the possibilities of its usage. The more flexible solution is to provide an API library, which would support all needed functionality, i.e. sending and receiving messages from different sources and to different destinations, parsing/generating them according to provided configuration and locally-defined messages, and providing the connectivity to a database as a source and storage for the data of processed messages.

Created HL7API library supports all of the functionality presented above. More over, it is possible to use only necessary functionality, or even to extend it in an easy way. Thus, the HL7API library can be used in the more variety of softwares.

3.1.2 HL7API library internal design

HL7API subsystems The HL7API library is internally divided into 3 logical subsystems:

- **Messages creation rules** — Concerns the **messages configuration**: configuration of all messages, segments, and fields stored in external files. Contains modules to access the configuration (**messages configuration reader** and **messages configuration writer**) and sets of internal representation buffers for storing it and providing it to message interpreter and generator.

- **Parsing and generating messages** — Main modules are **message interpreter** and **message generator**. Additional parts concern sending (**messages writer** with **ready-to-send messages buffer**) and receiving messages (**messages reader** with **received messages buffer**) to/from several messages destinations/sources, i.e.

TCP connections, simple *txt* files, *xml* files, as well as user-defined and user-implemented.

- **Database connectivity** — Concerns **database connectivity configuration**, configuration of databases and mappings of fields of segments and messages to a specific relations and/or columns in a relational database. Contains modules to access the configuration (**database connectivity configuration reader** and **database connectivity configuration writer**), access data in database (**database reader** and **database writer**), and sets of internal representation buffers for storing the configuration and providing it to database reader and database writer.

Internal structure and interactions between components of a subsystem and between components of different subsystems is shown in Figure 3.1. White rectangles are concrete components existing within HL7API, while shaded ones should be stored in the user-defined manner. Rectangles with rounded corners represent external elements of HL7API, like files, databases, and connections.

HL7API subsystems' components

Below general functionality of each of the components of presented subsystems as well as external interaction elements, is given.

Components of messages creation rules subsystem

- **Messages configuration reader** — Its responsability is to read *messages configuration* and store it in the *sets of internal representation of messages configuration*.
- **Messages configuration writer** — Its responsability is to write internally stored messages configuration into files which create the *messages configuration*.
- **Sets of internal representation of messages configuration** — Sets of buffer responsible for storing read *messages configuration* internally in the system. It provides the access to this data to *message interpreter* and *message generator* of *Parsing and generating messages subsystem*.

  Warning

  This is not a concrete component.

- **Messages configuration** — Set of files containing messages configuration. It contains files describing all defined fields, segments, and messages, as well as tables.

  Warning

  This is not a concrete component. This is an externally stored set of files.

Components of parsing and generating messages subsystem

- **Messages reader** — This component is responsible for reading/receiving messages from different types of sources, such as: *TCP connections*, *txt* files, and *xml* files. It is possible to easily extend its functionality to serve other types of message sources (i.e. databases). Read/received messages are stored in *received messages buffer*.
- **Messages writer** — This component is responsible for writing/sending generated messages to different types of destinations, such as: *TCP connections*, *txt* files, and *xml* files. It is possible to easily extend its functionality to serve other types of message destinations (i.e. databases). Messages for writting/sending are taken from *ready-to-send messages buffer*.

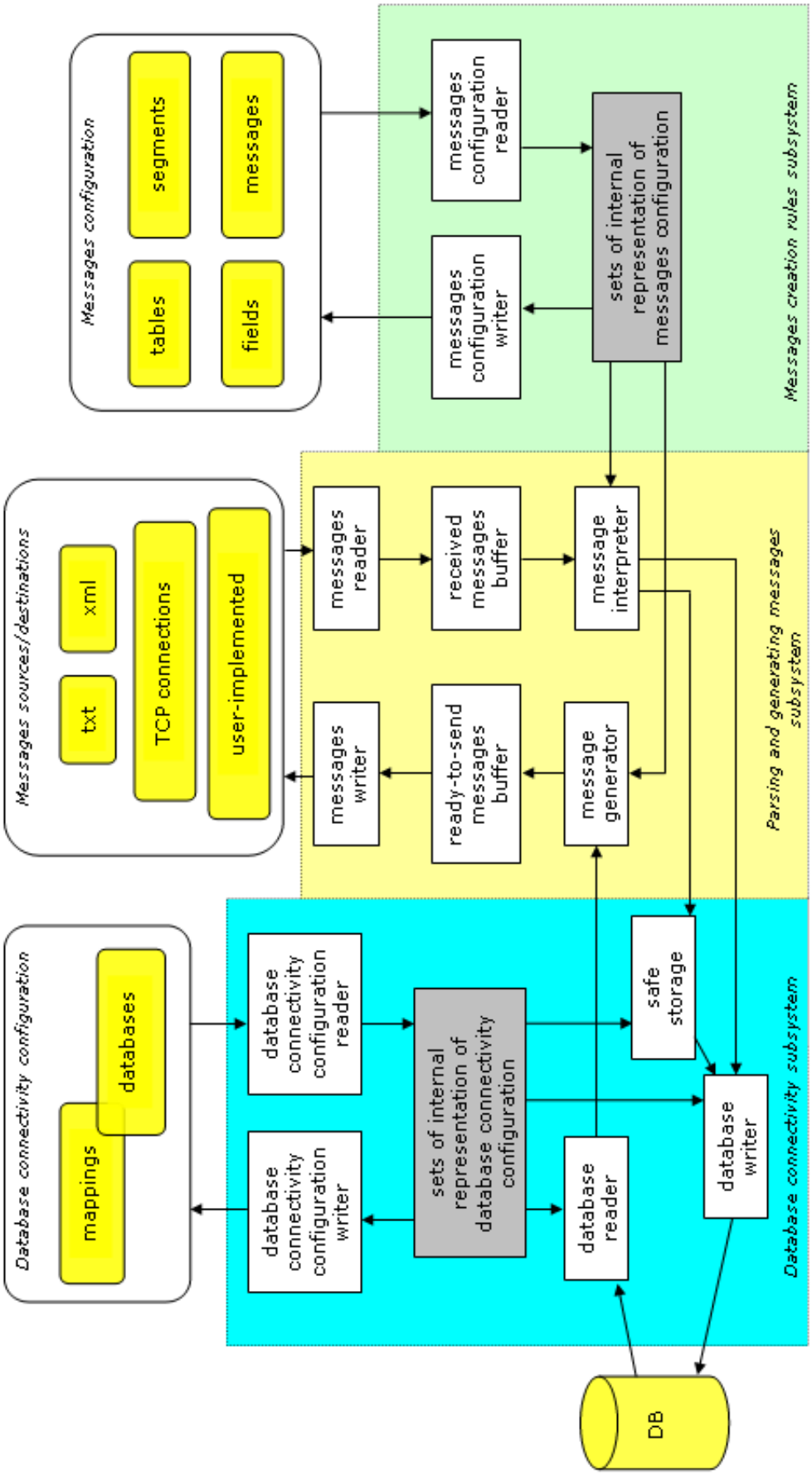


Figure 3.1 *HL7API library design*

- **Received messages buffer** — Serves as a buffer for messages received by *messages reader*.
- **Ready-to-send messages buffer** — Serves as a buffer for generated messages which are ready to be send/written to theirs destinations by *messages writer*.
- **Message interpreter** — Interprets received messages which are taken from *received messages buffer*. Type of message, its version and structural correctness are being checked with the use of internal representation of *messages configuration*, provided by *Messages creation rules subsystem*.
- **Message generator** — Generates messages from provided data and stores them in *ready-to-send messages buffer*. Messages are being generated according to the internal representation of *messages configuration*, provided by *Messages creation rules subsystem*.
- **Messages sources/destinations** — Different types of messages sources and destinations, such as *TCP connections*, *txt* files, or *xml* files. It is possible to use different sources/destinations of messages, if user provide appropriate reader/writer for the source/destination type.

> **Warning**
>
> This is not a concrete component. This is an external set of files, TCP connections, databases, etc.

Components of database connectivity subsystem

- **Database connectivity configuration reader** — Its responsability is to read *database connectivity configuration* and store it in the *sets of internal representation of database connectivity configuration*.
- **Database connectivity configuration writer** — Its responsability is to write internally stored database connectivity configuration into files which create the *database connectivity configuration*.
- **Sets of internal representation of database connectivity configuration** — Sets of buffer responsible for storing read *database connectivity configuration* internally in the system. It provides the access to this data to *database reader* and *database writer* of the same subsystem.

> **Warning**
>
> This is not a concrete component.

- **Database reader** — Its responsability is to read data from relational database as defined in internally stored *database connectivity configuration* on demand of *message generator* of *Parsing and generating messages subsystem*.
- **Database writer** — Its responsability is to write data from parsed messages to a relational database as defined in internally stored *database connectivity configuration* on demand of *message interpreter* of *Parsing and generating messages subsystem*.
- **Safe storage** — Its responsability is to commit data of incoming messages which suppose to be interpreted later to provided safe-storage system, such as database. Safe-storing data functionality is needed by a special mode of work of the *message interpreter* of *Parsing and generating messages subsystem*.
- **Database connectivity configuration** — Set of files containing database connectivity configuration. It contains files describing all used databases, as well

as mappings between all fields of all messages and columns in appropriate relations.

> **Warning**
>
> This is not a concrete component. This is an externally stored set of files.

- **Database (DB) —** Database or databases used by *database reader* and *database writer*.

> **Warning**
>
> This is not a concrete component. This is an external database or databases.

3.2 Messages configuration representation

All configuration files described below are *xml* files. For each of the files, DTD (see: [Document Type Definition]) is defined, presented in the form of a tree, and explained. All attributes are obligatory; if attribute can be optional it is stated in the text. If attribute is assigned a default value, it is stated, and the default value is provided as well.

3.2.1 Tables

Table configuration files

HL7 Tables described in Section 2.2.6 store default values for some of the fields. The HL7 Standard defines 2 types of tables: Standard-defined and user-defined. Structure of these tables do not vary, thus the representation is the same in both cases.

Data about each of the table is stored in separate file. Figure 3.2 shows the DTD tree for table configuration file. Each table, denoted here as TABLE, has the following 3 attributes:

- **NUMBER —** Gives the number of the table.

> **Warning**
>
> The HL7 Standard defines table numbers as 4 digits numbers, i.e. table number 120 is presented as "0120". In the presented configuration NUMBER is limited to significant digits only, i.e. "120" instead of "0120".

- **TYPE —** Type of the table, which can be "HL7" for Standard-defined tables or "User" for user-defined tables.
- **NAME —** Descriptive name of the table.

Table definition can be empty or contain non-limited number of ITEM elements. ITEM element has the following 2 attributes:

- **VALUE —** Predefined value.
- **DESCRIPTION —** Description of predefined value.

Examples of table configuration files can be found in Appendix B: Figure B.1 and Figure B.2.

Figure 3.2 Tree representation of DTD of table configuration file

```
TABLE (NUMBER, TYPE, NAME)
|
+= ITEM* (VALUE, DESCRIPTION)
```

Table index configuration file

Apart from table configuration files another file containing **table index** structure exists. DTD tree representation for table index file is presented on Figure 3.3. Table index, denoted here as TABLE_INDEX, contains no attributes. It's definition can be empty or contain non-limited number of TABLE entries, which has the following 2 atrributes:

- **NUMBER —** Number of the table.
- **FILE —** Filename, optionally with absolute or relative path to the file, in which table of number NUMBER can be found.

Example of table index configuration file can be found in Appendix B on Figure B.3.

Figure 3.3 Tree representation of DTD of table index configuration file

```
TABLE_INDEX
|
+= TABLE* (NUMBER, FILE)
```

3.2.2 Fields

Fields configuration file

Fields defined in Section 2.2.3 are the basic building parts of the messages. The HL7 Standard defines over 1000 fields used in different segments. Information about them is stored in one file, called **fields configuration file**, which contains all defined by the Standard fields, as well as can contain fields defined by the user.

Figure 3.4 Tree representation of DTD of fields configuration file

```
FIELDS
|
+- FIELD_DATA* (SEQ, NAME, ITEM, SEG, CHP?, LEN, DT, REP, QTY?,
   |            TABLE?, REQ, TYPE, DESCRIPTION?)
   |
   +- CM_COMPONENT* (SEQ, NAME, TYPE, REQ)
      |
      += CM_SUB_COMPONENT* (SEQ, NAME, TYPE, REQ)
```

DTD tree for fields configuration file is presented on Figure 3.4. There is one main element called FIELDS. It consists of 0 or more elements FIELD_DATA, each of which represents one field. Each field, denoted here as FIELD_DATA, has the following 13 attributes:

- **SEQ —** Sequence number or position within the specified segment.
- **NAME —** Descriptive name of the field.
- **ITEM —** Number uniquely identifying the field within the whole HL7 Standard. Fields added by the user must have this attribute set, and it has to be unique.
- **SEG —** Segment ID identifying segment which this field belongs to.
- **CHP —** Number or name of the chapter in which this field is defined in the Standard. This attribute is optional.

- **LEN —** Maximum length of the data of the field.
- **DT —** Data type of the field. Details: see Section 2.2.5.
- **REP —** Whether the field can repeat or not. Details: see Table 2.1, Repetition.
- **QTY —** Maximum number of times of field repetition. This attribute is optional. If not defined, field can repeat unlimited number of times.
- **TABLE —** Number or numbers of HL7 tables which has to be used to validate the value of the field (see also: Note: TABLE code). This attribute is optional. If it is not defined, field value validation will not take place.
- **REQ —** Whether the field is required or not. Details: see Table 2.1, Optionality.
- **TYPE —** Whether the field is defined by the HL7 Standard, or by the user.
- **DESCRIPTION —** Additional description of the field. This attribute is optional.

Note: TABLE code

It is important to notice that there are 2 principles of validating the field value. First, is being conducted independently of the configuration. It is defined in the HL7 Standard for several data types. Second, is the user-definable by providing appropraite table numbers for the fields.

In some special cases not the whole value of the field has to be validated with the values of the appropriate HL7 Table, but only a subcomponent of a field. Thus, a value of TABLE is a coded string. The string is constructed as follows:

- place a table number for component number 1 or leave blank if table not defined; put a comma ",",
- place a table number for component number 2 or leave blank if table not defined; put a comma ",",
- ...
- place a table number for component number *n* or leave blank if table not defined.

It is possible to omit spare commas, if omitting them does not cause loosing information about tables for components.

There is no possibility of defining a HL7 Table number to check the value of the subcomponent of the field.

Example. Field consists of 6 components. The value of second component should be checked with the values defined in the table number 100, and the value of fourth component with the table 101. All TABLE code presented in Figure 3.5 are valid.

Figure 3.5 Example TABLE codes

```
,100,,101,,
,100,,101,
,100,,101
```

CM components

In the case when field is of data type CM, it can contain components, denoted here as CM_COMPONENT elements. These elements define the structure of the CM data field, which can be freely defined. Each of them has the following 4 attributes:

- **SEQ —** Sequence number or position within the specified field.
- **NAME —** Descriptive name of the component of the CM data typed field.
- **TYPE —** Data type of the component.
- **REQ —** Whether the component is required or not. Values are limited to: "R" - required, and "O" - optional.

CM subcomponent

It is also possible for the CM_COMPONENT to have a data type CM. In this case, another level, CM_SUB_COMPONENT, is defined. Each of the CM_SUB_COMPONENT elements has the attributes defined as CM_COMPONENT element. The only limit is given for TYPE (data type) which can be only set to datatypes without components, i.e. DT, FT, ID, IS, NM, SI, ST, TM, TN, TS, or TX.

Example of fields configuration file can be found in Appendix B on Figure B.4.

3.2.3 Segments

Segments configuration file

Segments defined in Section 2.2.2 are the logical groupings of fields. The HL7 Standard defines approximately 100 different segments. Information about them is stored in one file, called **segments configuration file**, which contains all defined by the Standard segments, as well as can contain segments defined by the user.

Figure 3.6 Tree representation of DTD of segments configuration file

```
SEGMENTS
|
+= SEGMENT_DATA* (SEGMENT, DESCRIPTION, CHAPTER?, TYPE)
```

DTD tree for segments configuration file is presented on Figure 3.6. There is one main element called SEGMENTS. It consists of 0 or more elements SEGMENT_DATA, each of which represents one segment. Each segment, denoted here as SEGMENT_DATA, has the following 4 attributes:

- **SEGMENT —** Segment ID: 3-character unique identifier.
- **DESCRIPTION —** Description of the segment.
- **CHAPTER —** Number or name of the chapter in which this segment is defined in the Standard. This attribute is optional.
- **TYPE —** Whether the segment is defined by the HL7 Standard, or by the user.

Example of segments configuration file can be found in Appendix B on Figure B.5.

3.2.4 Messages

Messages configuration file

Messages defined in Section 2.2.1 are exchanged between participating systems. The HL7 Standard defines approximately 80 different types of messages. Information about them is stored in one file, called **messages configuration file**, which contains all defined by the Standard messages, as well as can contain messages defined by the user.

Single- and multi-version messages

All messages of the HL7 Standard have well defined structure. Nevertheless, the Standard allows for changing the structure of a message of a certain type under some circumstances. Even though the message has a different structure, it is still the message of the same type. Thus, we can talk of different versions of the same message. Such messages will be called **multi-version messages**; they are recognized during interpretation and/or generation by the specific data that they contain. The messages which structure is constant will be called **single-version messages**. Both types of messages are being stored in the same file.

DTD tree for mesages configuration file is presented on Figure 3.7. There is one main element called MESSAGES. It consists of 0 or more elements MESSAGE_DATA, each of which represents one message. Each message, denoted here as MESSAGE_DATA, has the following 4 attributes:

- **MESSAGE —** Message type: 3-character unique identifier.
- **DESCRIPTION —** Description of the message.
- **CHAPTER —** Number or name of the chapter in which this message is defined in the Standard. This attribute is optional.
- **TYPE —** Whether the message is defined by the HL7 Standard, or by the user.

Apart from that, 3 elements exist: SEGMENTS_DEF, COMMON_START_SEGMENTS, and VERSION_SEGMENTS. When defining single-version message only SEGMENTS_DEF is used, while for definition of multi-version message all above elements are used.

Figure 3.7 Tree representation of DTD of messages configuration file

```
MESSAGES
|
+- MESSAGE_DATA* (MESSAGE, DESCRIPTION, CHAPTER?, TYPE)
   |
   |- SEGMENTS_DEF
   |  |
   |  |= SEGMENT (SEQ, SEG, OPTIONAL, REPEAT, VALID="Yes")
   |  |
   |  +- GROUP_OF_SEGMENTS (NAME, OPTIONAL, REPEAT, REPEAT_QTY?,
   |     |                   DESCRIPTION?)
   |     |
   |     +- SEGMENTS_DEF  **
   |
   |- COMMON_START_SEGMENTS
   |  |
   |  +- SEGMENTS_DEF  -->
   |
   +- VERSION_SEGMENTS
      |
      +- VERSION+ (NAME, DESCRIPTION?, DEFAULT?)
      |
      |- CONDITION
      |  |
      |  += CND_DEF+ (CND_SEQ, SEG_ID, FIELD_SEQUENCE,
      |               FIELD_COMPONENT_NUMBER?,
      |               FIELD_SUB_COMPONENT_NUMBER?, OPERAND, VALUE,
      |               CONNECTION_OPERATOR?)
      |
      +- SEGMENTS_DEF  -->
```

Single- and multi-version message elements

SEGMENT_DEF element

Element SEGMENTS_DEF consists of 0 or more SEGMENT and/or GROUP_OF_SEGMENTS elements.

SEGMENT element

Element SEGMENT describes the segment of message. Each of the segments has the following 5 attributes:

- **SEQ —** Sequence number or position of the segment within a message.
- **SEG —** Segment ID.
- **OPTIONAL —** Whether the segment is optional or not. Values limited only to: "No" - required, and "Yes" - optional.
- **REPEAT —** Whether the segment is allowed to repeat or not. Values limited only to: "No" - cannot repeat, and "Yes" - can repeat.
- **VALID —** Whether the segment is a valid segment. Unfortunatelly, in message SUR one of the segments is not a valid segment, but a separate field of type ED. Values limited only to: "Yes" - valid segment, and "No" - invalid segment. This attribute, if not present, gets the default value of "Yes".

GROUP_OF_SEGMENTS element

The GROUP_OF_SEGMENTS element is defined as a SEGMENT_DEF element, which means it can contain SEGMENT elements and/or GROUP_OF_SEGMENTS elements. The GROUP_OF_SEGMENTS element has the following 5 attributes:

- **NAME —** Descriptive name of the group of segments.
- **OPTIONAL —** Whether the group of segments is optional or not. Values limited only to: "No" - required, and "Yes" - optional.
- **REPEAT —** Whether the group of segments is allowed to repeat or not. Values limited only to: "No" - cannot repeat, and "Yes" - can repeat.
- **REPEAT_QTY —** Maximum number of times which the group of segments is allowed to repeat. This attribute is optional. If not present, the group of segments can repeat unlimited number of times.
- **DESCRIPTION —** Additional description of the message. This attribute is optional.

Single-version message

The single-version message consists of one element SEGMENTS_DEF.

Example of single-version message definition can be found in Appendix B on Figure B.6.

Multi-version message

The multi-version message consists of one COMMON_START_SEGMENTS followed by one VERSION_SEGMENTS element. The details follow.

COMMON_START_SEGMENTS element

The COMMON_START_SEGMENTS element is defined as one element of SEGMENTS_DEF. It has no attributes.

The idea of this element is to group all the segments and/or group of segments common for all the message versions that appear at the beginning of the message structure. All following segments and/or group of segments differ throughout the versions. The message version have to be recognized using data that appear in these segments.

As all messages are being stored in the same file, existance of this element in the message definition determines the message for being a multi-version message.

VERSION_SEGMENTS element

The VERSION_SEGMENTS element consists of 1 or more VERSION elements. VERSION_SEGMENT has no attributes.

This element stores all the data related to different versions of the multi-version message.

VERSION element

The VERSION element defines the version of the message, providing information about condition (or conditions) that have to be met for using a certain message structure (element

CONDITION), and about the structure of the version of the message (element SEGMENTS_DEF).

> **Warning**
>
> SEQ in the segments appearing in all of the VERSION elements has to start from the number one greater than the last SEQ of the segment of COMMON_START_BRICKS.

VERSION element has the following 3 attributes:

- **NAME —** The name (identifier) of the message version.
- **DESCRIPTION —** Description of the message version. This attribute is optional.
- **DEFAULT —** Whether this message version is the default version of the message in case of lack of the possibility of recognizing the correct version. Values limited to: "Yes", and "No". If more than one versions are set to be default, the behaviour of interpreter and/or generator is undetermined. This attribute is optional.

CONDITION element The VERSION element contains one CONDITION element. This condition is given to the value or values of the specified field, component of the field, or subcomponent of the field of a segment contained in COMMON_START_SEGMENTS element. If the condition is met, the version of the message is being determined. CONDITION element has 0 attributes, but it consists of one or more CND_DEF elements, called condition bricks. CND_DEF is the brick of the condition. It has the following 8 attributes:

- **CND_SEQ —** Sequence number or position of the CND_DEF element within CONDITION element.
- **SEG_ID —** ID of the segment from COMMON_START_SEGMENTS which contains the field which the condition brick has to be applied to.
- **FIELD_SEQUENCE —** Sequence number or position of the field of specified segment which the condition brick has to be applied to.
- **FIELD_COMPONENT_NUMBER —** Number of the component of the field of the specified segment which the condition brick has to be applied to. This attribute is optional. Nevertheless, if it is specified, the condition brick has to be applied to the value of the component of the field, not to the value of the field.
- **FIELD_SUB_COMPONENT_NUMBER —** Number of subcomponent of the component of the field of the specified segment which the condtion brick has to be applied to. This atrribute is optional. Nevertheless, if it is specified, the condition brick has to be applied to the value of the subcomponent of the component of the field, not to the value of the component of the field nor to the value of the field. Moreover, if this atrtribute is specified, FIELD_COMPONENT_NUMBER is not optional.
- **OPERAND —** Comparision operator which has to be used while comparing specified value with the defined field, component of the field, or subcomponent of the component of the field of the defined segment. Values limited to: "EQUAL" (==), "LESS" (<), "GREATER" (>), "LESS_EQUAL" (<=), "GREATER_EQUAL" (>=), and "DIFFERENT" (!=).
- **VALUE —** Value which should be compared to the defined field, component of the field, or subcomponent of the component of the field of the defined segment.
- **CONNECTION_OPERATOR —** Operator used for connecting the result of condition brick (true or false) with the next condition brick (if defined). Values limited to: "AND", and "OR". This attribute is optional. Nevertheless, when more

than one condition bricks are defined, this attribute is optional only for the last (ordered by CND_SEQ) of the condition bricks.

Example of multi-version message definition can be found in Appendix B on Figure B.7.

3.3 Interpreter internal configuration

Messages interpreter is one of the most crucial parts of the HL7API library. Thus, providing it with the correct configuration is very important. The configuration of messages parser is presented below.

3.3.1 Interpreter configurtation overview

Configuration of the message interpreter can be divided into 2 groups:

- **Message validity** — Configuration allowing to check whether incoming message is of proper type, and whether it follows the defined for this type of message structure.
- **Message treatment** — Configuration allowing performing some operations after and/or before interpreting a message.

Details are provided below.

3.3.2 Message validity configuration

Needed information

For the validity of message checking, data concerning structure of messages has to be provided to the interpreter. It can be easily done with the use of *Messages creation rules subsystem*. It provides all data concerning structure of a message, its segments, fields, and tables containing default values for the field data.

Validation levels

Three levels of validation for a message are being defined:

1. **Field level** — Validation of stored data, if it follows the order and structure defined for a certain data type, as well as validation of the value of the field, or its component with the predefined values of certain HL7 tables.

   Warning

   Testing value of the field or its component on the field level follows the rules defined in the HL7 Standard for field construction. It is impossible to switch it off with the use of configuration files.

2. **Segment level** — Whether segments contain proper number of fields, in correct order, and of correct type. Also, if user defines the number of table from which the field should have a value, it is being tested on the segment level for each of the fields.
3. **Message level** — Whether the message contains correct segments in correct positions, and if the required segments and/or group of segments are present.

Field level validation

Validation of the *field level* takes place during data interpretation. First, before setting the data, it is being tested if it follows the rules of the specified data type. If yes, then its value is tested with the values defined in the appropriate HL7 table.

> **Warning**
>
> Testing the field value on the field level is independent on the configuration and cannot be switched off.

Segment level validation Segment level validation is conducted to all of the segments that are being constructed. First, it is checked if the segment with the given segment ID exists within the configuration. Next, if the number of fields of the constructed segment is correct. Lastly, if the data types of the fields are correct and in correct order, and, if user defines additional HL7 tables numbers, validity of the contained data.

Message level validation Message segment validation is being conducted after constructing all the segments of the message. At the beginning it is checked if the message with the given message type or ID exists in the configuration. Following, correct order, repetition, and optionality of the segments is checked.

3.3.3 Message treatment configuration

ACK modes Messages interpreter can work in 2 modes of intepretation: **original** and **enhanced**. In *original* mode, the incoming message is interpreted immediately. In the *enhanced* mode the message is firstly commited to the safe storage, and then interpreted immediately or later.

Safe storage If the mode of interpreter is set to *enhanced* mode, it has to be provided with the appropriate linker to the *safe storage* object (with the use of *Database connectivity subsystem*), which allows the interpreter to commit the data of the message. Moreover, it is possible to only safe-store the data without interpreting them, or do both actions.

Database storing After interpreting the message it is possible to automatically store the interpreted data in the database(s). For doing this action an appropriate object from *Database connectivity subsystem* has to be configured and provided to the interpreter (see: Section 3.5).

3.4 Generator internal configuration

Messages generator is second of the most crucial parts of the HL7API library. Thus, providing it with the correct configuration is very important. Below the configuration of messages generator is presented.

3.4.1 Generator configuration overview

Configuration of message generator is very similar to configuration of message interpreter (described in Section 3.3), and can also be divided into 2 subgroups:

- **Message construction** — Configuration allowing to determine the structure of the generated message.
- **Message data source** — Configuration allowing retrievieng data for the generated message directly from the database.

Details are provided below.

3.4.2 Message construction

The messages are constructed according to the configuration provided by the *Messages creation rules subsystem*. The validity of them is also being checked while construction in the manner analogous to the one described in Section 3.3.

3.4.3 Message data source

It is possible for the message generator to generate messages using the data stored in the database(s). For doing this an appropriate object from *Database connectivity subsystem* has to be configured and provided to the message generator (see: Section 3.5).

3.5 Database connectivity configuration

Database connectivity configuration is needed by the components of *Database connectivity subsystem* to perform operations on database(s). The configuration consists of description of available database(s), relations defined in the database(s), and mappings between fields of certain segment of certain message and the columns of the specified relation in the specified database. There is one special relation considered as the relation for use in safe-storaging the messages, if the interpreter works in *enhanced* mode (see: Section 3.3.3).

All described below configuration files are *xml* files. For each of the files, DTD is defined, presented in the form of a tree, and explained. All attributes are obligatory; if attribute can be optional it is stated in the text. If attribute is assigned a default value, it is stated, and the default value is provided as well.

3.5.1 Databases

Databases configuration file

The *Database connectivity subsystem* is able to provide connection to database(s). For this, it needs information of the databases which are available in the system for the usage. This information is stored in the **databases configuration file**.

DTD tree for databases configuration file is presented on Figure 3.8. There is one main element called DATABASES. It consists of 0 or more DATABASE elements, each of which represents one database configuration. Each database, denoted here as DATABASE, has the following 3 attributes:

- **ID** — Unique identifier of the database throughout the HL7API library.
- **TYPE** — Type of the mode of access to the database. Currently, values are limited to: "OCI" — dedicated *Oracle Call Interaface* for the use with Oracle database, and "ODBC" for other databases. (For OCI, see: [OCI Web page]; for Oracle, see: [Oracle Web page]).
- **DESCRIPTION** — Description of the database. This attribute is optional.

Figure 3.8 *Tree representation of DTD of databases configuration file*

```
DATABASES
|
+- DATABASE* (ID, TYPE, DESCRIPTION?)
   |
   |= DB_OCI (USER, PASS, DBLINK?)
   |
   += DB_ODBC (DATA_SOURCE_NAME)
```

Each DATABASE element has defined as its child one DB_OCI element, or one DB_ODBC element, providing all needed configuration parameters for making a connection with the specified database type.

DB_OCI element

The DB_OCI element provides parameters of connection for the OCI-typed database. It contains the following 3 attributes:

- **USER —** Name of the user of the database.
- **PASS —** Password of the defined user of the database.

> **Warning**
>
> Special care and/or special measures has to be takes, as in current version password of the user is stored as plain text only.

- **DBLINK —** Name of the database link which has to be used for creating connections. This attribute is optional. If the attribute is empty, the connections will be made to the default instance of the database.

DB_ODBC element The DB_ODBC element provides parameters of connection for the ODBC-typed database. It contains the following 1 attribute:

- **DATA_SOURCE_NAME —** Name of the defined ODBC data source.

Example of databases configuration file can be found in Appendix B on Figure B.8.

3.5.2 Relations

Relations configuration file All operations which are made by the *Database connectivity subsystem* in databases has to be made on some relation(s). For this, it is needed to know the available relation(s) in defined database(s). This information is stored in the **relations configuration file**.

DTD tree for database relations configuration file is presented on Figure 3.9. There is one main element called RELATIONS. It consists of 0 or more RELATION elements, each of which represents one database relation configuration. Each database relation, denoted here as RELATION, has the following 4 attributes:

- **ID —** Unique identifier of the relation throughout the HL7API library.
- **DATABASE —** Identifier of the database defined in the databases configuration file (attribute: ID).
- **NAME —** Name of the relation as it appears in the database.
- **DESCRIPTION —** Description of the relation. This attribute is optional.

Figure 3.9 *Tree representation of DTD of relations configuration file*

```
RELATIONS
|
+= RELATION* (ID, DATABASE, NAME, DESCRIPTION?)
```

Example of relations configuration file can be found in Appendix B on Figure B.9.

3.5.3 Mappings

Mappings configuration file For the automatical storing of interpreted messages (see: Section 3.3.3) and/or generating messages taking data directly from the database (see: Section 3.4.3), mappings between fields of the specified segments of the specified messages and column name of the specified relation in the specified database have to be defined. These mappings are stored in the **mappings configuration file**.

Mapping definition is divided into 2 steps: firstly, the mapping between each of the segments of the message or message version (if exists) is defined; secondly, detail mappings between fields of the defined segment of the message can be defined.

DTD tree for mappings configuration file is presented on Figure 3.10. There is one main element called MAPPINGS. It consists of 0 or more MAPPING elements, each of which represents one mapping between specified segment of a specified version (if exists) of a specified message. Each mapping, denoted here as MAPPING, has the following 6 attributes:

- **MESSAGE —** Identifier of the message.
- **VERSION —** Name of the version of the multi-version message. This attribute is optional only if the stored data concerns single-version message. In other case, it is required for proper functioning.
- **SEGMENT —** Identifier of the segment.
- **SEGMENT_POSITION —** Segment position within the message, as defined in messages configuration file. This field is crucial, as segments of the same identifier can appear in the message in different positions concerning different data.
- **DEFAULT_DBRELATION —** Identifier of the database relation defined in the database relations configuration file (attribute: ID). This attribute is optional. For more deatails of usage, see: "Database relations terms of use" at the end of this subsection.
- **DESCRIPTION —** Description of the mapping. This attribute is optional.

Figure 3.10 Tree representation of DTD of mappings configuration file

```
MAPPINGS
|
+- MAPPING* (MESSAGE, VERSION?, SEGMENT, SEGMENT_POSITION,
   |          DEFAULT_DBRELATION?, DESCRIPTION?)
   |
   += MAPPING_ITEM (DBRELATION?, ITEM, COLUMN?, DESCRIPTION?)
```

MAPPING_ITEM element

Additionally, each of the MAPPING elements can contain 0 or more MAPPING_ITEM elements, which specifies the mapping between each of the fields and the column of the relation in the database. The MAPPING_ITEM element has the following 4 attributes:

- **DBRELATION —** Identifier of the database relation defined in the database relations configuration file (attribute: ID). This attribute is optional. For more details of usage, see: "Database relations terms of use" at the end of this subsection.
- **ITEM —** Number of the field of the segment.
- **COLUMN —** Name of the column of the database relation. This attribute is optional. See also: "Database relations terms of use" at the end of this subsection.
- **DESCRIPTION —** Description of the mapping. This attribute is optional..

Example of mappings configuration file can be found in Appendix B on Figure B.10.

> **Database relations terms of use**
>
> There are 3 possible levels of defining the mappings for the message data and columns:
>
> 1. **Segment level —** Allowing for storing all the fields of a defined segment in one database relation.
> 2. **Field level —** Allowing to define the column and the database relation separately for all of the fields of the segment.

3. **Mixed level** — Allowing to use above ways in the same moment.

Segment level mappings

Segment level mapping allows to drastically limit the amount of defined MAPPING_ITEM elements, as it allows to define only a mapping between a specified segment on a specified position of the message and the database relation. In such case, the DEFAULT_DBRELATION of the MAPPING element has to be defined and is used as a relation for storing the message segment. Element MAPPING contain no MAPPING_ITEM elements. Name of columns for all the fields of the message are generated automatically in the following fashion:

```
FLD_xx
```

where *xx* is the number of the field within the segment without the leading 0 (zero).

Field level mappings

Field level mappings are the most detailed mappings, as they allow to define a separate database, relation, and column for each of the fields of the message segment. In such case, DEFAULT_DBRELATION element of MAPPING element is optional, but MAPPING element has to have defined MAPPING_ITEM elements for all of the fields of the segment. The DBRELATION attribute of each of the MAPPING_ITEM elements is required and has to be set as well.

Mixed level mappings

Mixed mappings allow the user to define both the segment and field level mappings for the same message. Used database relation and column name differs, depending on data provided by the user, and are presented in Table 3.1.

Table 3.1 *Mixed level mappings relation and column name usage*

Provided by user			Used	
DEFAULT_ DBRELATION	**DBRELATION**	**COLUMN**	**Relation**	**Column name**
Yes	Yes	Yes	DBRELATION	COLUMN
Yes	Yes	No	DBRELATION	*FLD_xx*
Yes	No	Yes	DEFAULT_ DBRELATION	COLUMN
Yes	No	No	DEFAULT_ DEBRELATION	*FLD_xx*
No	Yes	Yes	DBRELATION	COLUMN
No	Yes	No	DBRELATION	*FLD_xx*
No	No	Yes	*error*	*error*
No	No	No	*error*	*error*

3.5.4 Safe-storage

Safe-storage configuration file

The message interpreter for its proper functioning may need a safe-storage functionality (see: Section 3.3.3). Thus, a special relation for safe-storing messages is needed. The information about this relation is stored in the **safe-storage relation configuration file**.

DTD tree for safe-storage relation configuration file is presented on Figure 3.11. There is one main element called SAFE_STORAGE_RELATION. It consists of 1 SSRELATION element,

which represents one database relation configuration used for safe-storing the messages. The SSRELATION has the following 4 attributes:

- **DATABASE —** Identifier of the database defined in the databases configuration file (attribute: ID).
- **NAME —** Name of the relation as it appears in the database.
- **COLUMN —** Name of the column of the database relation.
- **DESCRIPTION —** Description of the safe-storage relation. This attribute is optional.

```
SAFE_STORAGE_RELATION
|
+= SSRELATION (DATABASE, NAME, COLUMN, DESCRIPTION?)
```

Figure 3.11 *Tree representation of DTD of safe-storage relation configuration file*

Example of safe-storage relation configuration file can be found in Appendix B on Figure B.11.

4 Implementation

This section provides detailed information concerning selected implementation issues and limitations. In the first section, the developing environment as well as the target environment of the HL7API is presented. Following sections gives information about used 3rd-party sources and libraries. Then, details concerning developed algorithms used in the process of validation of the structure of the message and automated generation of them are presented. Lastly, specific limitations of the structure of the database relations used during automatic storing and generating messages are presented.

4.1 Environment

Developing environment The developing environment consisted of Borland C++ Builder 5 with the Oracle8*i* Database Server. The basic database connection is being dedicated to Oracle databases, which are using OCI interface.

The HL7API library has been developed as a static linked library.

Target environment Basic taget environment is similar to the one described above: Borland C++ Builder 5 with the Oracle8*i* Database Server. Nevertheless, there are no limitations put on the internal structure of the library preventing of compiling sources as a dynamic linked library, nor using the sources directly in the developed application. Moreover, the User can implement connection routines to databases different the Oracle; there are prepared classes which can facilitate the developing process for ODBC-based conenctions. More features can also be added to all of the classes of the HL7API library.

More information concerning Borland, see: [Borland Web page], C++ Builder, see: [Borland C++ Builder Web page], Oracle, see: [Oracle Web page].

4.2 Other sources and libraries

OCI The Oracle Call Interface (OCI) is an application programming interface (API) that allows applications to interact with one or more Oracle servers. The OCI gives the capability to perform the full range of database operations that are possible with an Oracle Database Server.

The used version of the OCI library was provided with the Oracle8*i* Database Server as a statically linked library.

More information about Oracle databases can be found in [Oracle Web page]. More information about OCI can be found in [OCI Web page] and [OCI Programmer's Guide].

TinyXml TinyXml is a simple C++ XML parser. It reads XML and creates C++ objects representing the XML document. The objects can be manipulated, changed, and saved again as XML.

The used version of the TinyXml is 2.4.3. It comes as a set of source files. It was used without sources modifications.

More inforamtion about TinyXml can be found in [TinyXml Web page].

4.3 Algorithms

During message interpretation and generation not only the validity of some specified fields or its components has to be conducted (as described in Section 3.3.2, field level), but also the overall structure of the message has to be checked (defined in Section 3.3.2, message level).

Within the created and implemented algorithms, 2 groups of them can be distinguished: **parsing-time algorithm**, and **generation-time algorithm**. Both of these groups of algorithms are presented and described below. Moreover, one specialized algorithm used by the algorithms of the above groups is presented, too.

Parsing-time algorithms Parsing-time algorithms are used during or just after finishing the process of interpreting the message. The function of these algorithms is to check if the interpreted message has the structure as defined in the *Message configuration*. This kind of algorithm can also be used during the process of generation the message, to check if the generated message has valid structure.

Generation-time algoritm Generation-time algorithms are used in the process of automatic messages generation (see: Section 3.4). They are not used to validate the structure of the message, but to provide the order of construction of segments of a message.

Recursive algorithms The structure of a message and the *Messages configuration* structures are recursive (for reminding: a message is defined as ordered list of segments and/or group of segments, and the group of segments is constructed of an ordered list of segments and/or group of segments, and so on; for details see Section 3.2.4). Thus, the most natural way of processing this data is also recursive, as iterative processing could cause some crucial data loss (i.e. whether the segment is in the group, and how deep). For this reason all the algorithms presented here are recursive.

4.3.1 Current position in message

Determining a sequence number of the segment within a message is very easy. Nevertheless, this information is not enough for correct functioning of the algorithms presented below. Thus, a recursive navigating to the current position in the message is needed.

The algorithm of navigating through the message to the current position in it, NAVIGATE, is presented in Figure 4.1.

> **Important**
>
> For preserving all the information obtained by the NAVIGATE algorithm and for taking advantage of the recursive nature of this algorithm, it has to be integrated with the algorithms presented below.

Figure 4.1 *NAVIGATE algorithm*

Input

brick — the first element (*message-brick*) of the proper message (with determined version, if exists) configuration; it can only be a segment, as all messages have to start with MSH segment

position — position up to where to navigate

Output

brick (*message-brick*) up to which the algorithm has navigated, (possibly) with all the information concerning depth of the group(s) and its (theirs) optionality and possibility of repetition, as well as the position (sequence number within a message) of the returned segment

Stop condition

Algorithm stops in the case of:

- successfully navigating to the given position (lines: 13-14, and 16)
- returning an ERROR value (lines: 2, and 18)
- navigating until the end of the message without finding given position; detection of this state is denoted below as: *MessageFinished()* condition (line: 3)

```
NAVIGATE(brick, position)

1   if NULL = brick
2   then return ERROR

3   while not FinishedScanning() and not MessageFinished()

4      if Type(brick) = GROUP_OF_SEGMENTS
5         TreatBrickAsGroup(brick)
6         brick := GetFirstBrickOfGroup(brick)
7         NAVIGATE(brick, position)
8         if not FinishedScanning() then
9         brick := GetNextBrick(brick)

10     else if Type(brick) = SEGMENT
11        TreatBrickAsSegment(brick)
12        if position = 0
13        then position = 1
14             FinishedScanning()
15        else if position = SequenceOfSegment(brick)
16             then FinishedScanning()
17             else brick := GetNextBrick(brick)

18     else return ERROR
```

NAVIGATE algorithm notes

lines 1-2: check if the brick is a valid object

line 3: main loop of the algorithm, working until the message is fully scanned

lines 4-9: if the considered brick is a group, it has to be treated as a group (line 5); starting with the first brick of the considered group (line 6) launch NAVIGATE algorithm recuresively (line 7); after returning from recursive call, check if the scanning was not finished (line 8) and if not, get another brick (line 9)

lines 10-17: if the considered brick is a segment, it has to be treated as a segment (line 11); if the searched position is equal to 0 (zero) means, that this algorithm is called for the first time; thus the first element to which can navigate is of position equal to 1 (one) (lines: 12-14), which is a valid message position, and the algorithm is finishing the work; else check if the sequence number of the segment (position within a message) is equal to the searched position (line 15); if yes, algorithm is finishing the work (line 16); otherwise, get next brick of the message (line 17)

line 18: return an ERROR if the brick type is not recognized

4.3.2 Segment position

After interpreting the message, which comes as a string of characters, an object representing the message is created. Even though this message and its segments follows the rules of message creation, the information about the position of the segment is lost, as it is not a crucial in the point of view of a system processing data carried by the message. Thus, a position of the segment within a message has to be regenerated.

For regenerating the sequence number of the segment within the message, an algorithm SEGMENT_POSITION presented on Figure 4.2 is being used.

Figure 4.2 SEGMENT_POSITION algorithm

Input

brick — the first element (*message-brick*) of the proper message (with determined version, if exists) configuration; it can only be a segment, as all messages have to start with MSH segment

position — position of the last detected segment position, or 0, when no positions where detected up to now.

> **Warning**
>
> For correct functioning of this algorithm, feedback of the calling routine is necessary for this parameter. Thus, first call to SEGMENT_POSITION should be with the position equal to 0 (zero), while next with the position equal to the one which was returned in the previous call to this function for a specified message or message version (if defined).

segment ID — ID of the segment which position within a message should be determined.

Output

Position on which the desired segment has been found.

Stop condition

Algorithm stops in the case of:

- successfully determining the position of the searched segment (lines: 20, and 24)
- returning an ERROR value (lines: 2, and 26)
- navigating until the end of the message without finding the given segment; detection of this state is denoted below as: *MessageFinished()* condition (line: 4)

```
SEGMENT_POSITION(brick, position, segment_id)

1   if NULL = brick
2   then return ERROR

// Phase 1: navigate to appropriate position
3   NAVIGATE(brick, position)

// Phase 2: find a position of a provided segment
4   while not FinishedScanning() and not MessageFinished()

5      if Type(brick) = GROUP_OF_SEGMENTS
6         TreatBrickAsGroup(brick)
7         brick := GetFirstBrickOfGroup(brick)
8         SEGMENT_POSITION(brick, position, segment_id)
9         if not FinishedScanning() then
```

```
10            if GroupCanRepeat(brick) = false then return;
11            else if GroupWasScannedTwoTimes(brick) = false
12                    then GroupWasScannedTwoTimes(brick) := true
13                         ScanGroupAgain(brick)
14                 else return

15      else if Type(brick) = SEGMENT
16         TreatBrickAsSegment(brick)
17         if FirstFoundSegment(brick) = true
18            if SegmentID(brick) = segment_id
19               and SegmentCanRepeat(brick) = true
20               then FinishedScanning()
21               else brick := GetNextBrick(brick)
22         else
23            if SegmentID(brick) = segment_id
24            then FinishedScanning()
25            else brick := GetNextBrick(brick)

26      else return ERROR
```

SEGMENT_POSITION algorithm notes

lines 1-2: check if the brick is a valid object

line 3: navigate to current position in the message (see: Section 4.3.1);

> **Warning**
>
> This function is being called only once during the whole algorithm. In the recursive calls of SEGMENT_POSITION, call to NAVIGATE should be omitted.

line 4: main loop of the algorithm

lines 5-14: if the considered brick is a group of segments, it has to be treated as a group (line 6); starting with the first brick of the considered group (line 7) launch SEGMENT_POSITION algorithm recuresively (line 8), but starting from *phase 2* of the algorithm; after recursive call return, if not yet FinishedScanning() (line 9) check the following conditions:

- if group cannot repeat, finish the recursive call and return (line 10)
- otherwise, if group was not scanned to the end second time, scan the group again (lines 11-13)
- else finish the recursive call and return (line 14)

lines 15-25: if the considered brick is a segment, it has to be treated as a segment (line 16); two cases are considered:

- if this segemnt is the segment which was found for the first time (line 17) after the call to NAVIGATE in line 3, then if segment ID of the brick is equal to searched segment ID and the brick is allowed to repeat, then the searched segment has been found (lines 18-20); otherwise, get next brick and repeat procedure (line 21)
- in the other case, if the the considered segment is not the first segment found after NAVIGATE call (line 22), then if segment ID of the considered brick is equal to searched segment ID (line 23), the searched segment has been found (line 24); otherwise, get next brick and repeat procedure (line 25)

line 26: return an ERROR if the brick type is not recognized

4.3.3 Incoming segment validity

During the process of interpretation, an interpretet message is being built from the incoming data field by field and segment by segment. Validating the structure of the certain segment is not complicated. Nevertheless, as message have more sophisticated structure, a special algorithm, checking if the *incoming* segment is valid or not, is needed

Checking the validity of incoming segment, i.e. if the segment of known ID can be added at the end of the message being constructed without violating the structure of it, is done by the algorithm INCOMING_SEGMENT_VALID presented on Figure 4.3.

Figure 4.3 INCOMING_SEGMENT_VALID algorithm

Input

brick — the first element (*message-brick*) of the proper message (with determined version, if exists) configuration; it can only be a segment, as all messages have to start with MSH segment

position — position of the last detected segment position, or 0, when no positions where detected up to now.

> **Warning**
>
> For correct functioning of this algorithm, feedback of the calling routine is necessary for this parameter. Thus, first call to INCOMING_SEGMENT_VALID should be with the position equal to 0 (zero), while next with the position equal to the one which was returned in the previous call to this function for a specified message or message version (if defined).

segment ID — ID of the segment which suppose to be added at the end of message being constructed

Output

true or **false**, as well as the position of the new segment within a message

Stop condition

Algorithm stops in the case of:

- successfully determining whether the segment can be added to the message (lines: 20-21, and 25-26) or not (lines: 28-29, and 32)
- returning an ERROR value (lines: 2, and 31)
- navigating until the end of the message without finding the given segment; detection of this state is denoted below as: *MessageFinished()* condition (line: 4)

```
INCOMING_SEGMENT_VALID(brick, position, incoming_segment_id)

1   if NULL = brick
2   then return ERROR

// Phase 1: navigate to appropriate position
3   NAVIGATE(brick, position)

// Phase 2: find a position of a provided segment
4   while not FinishedScanning() and not MessageFinished()

5      if Type(brick) = GROUP_OF_SEGMENTS
```

```
 6          TreatBrickAsGroup(brick)
 7          brick := GetFirstBrickOfGroup(brick)
 8          INCOMING_SEGMENT_VALID(brick, position, incoming_segment_id)
 9          if not FinishedScanning() then
10             if GroupCanRepeat(brick) = false then return;
11             else if GroupWasScannedTwoTimes(brick) = false
12                     then GroupWasScannedTwoTimes(brick) := true
13                          ScanGroupAgain(brick)
14                  else return

15       else if Type(brick) = SEGMENT
16          TreatBrickAsSegment(brick)
17          if FirstFoundSegment(brick) = true
18             if SegmentID(brick) = incoming_segment_id
19                and SegmentCanRepeat(brick) = true
20                then FinishedScanning()
21                     return TRUE
22                else brick := GetNextBrick(brick)
23          else
24             if SegmentID(brick) = incoming_segment_id
25             then FinishedScanning()
26                  return TRUE
27             else if SegmentIsOptional(brick) == false
28                  then FinishedScanning()
29                       return FALSE
30                  else brick := GetNextBrick(brick)

31       else return ERROR

32    return FALSE
```

INCOMING_SEGMENT_VALID algorithm notes

lines 1-2: check if the brick is a valid object

line 3: navigate to current position in the message (see: Section 4.3.1);

> **Warning**
>
> This function is being called only once during the whole algorithm. In the recursive calls of INCOMING_SEGMENT_VALID, call to NAVIGATE should be omitted.

line 4: main loop of the algorithm

lines 5-14: if the considered brick is a group of segments, it has to be treated as a group (line 6); starting with the first brick of the considered group (line 7) launch INCOMING_SEGMENT_VALID algorithm recursively (line 8), but starting from *phase 2* of the algorithm; after recursive call return, if not yet FinishedScanning() (line 9) check the following conditions:

- if group cannot repeat, finish the recursive call and return (line 10)
- otherwise, if group was not scanned to the end second time, scan the group again (lines 11-13)
- else finish the recursive call and return (line 14)

lines 15-30: if the considered brick is a segment, it has to be treated as a segment (line 16); two cases are considered:

- if this segemnt is the segment which was found for the first time (line 17) after the call to NAVIGATE in line 3, then if segment ID of the brick is equal to searched segment ID and the brick is

allowed to repeat, then the searched segment can be validly added to the message (lines 18-21); otherwise, get next brick and repeat procedure (line 22)

- in the other case, if the the considered segment is not the first segment found after NAVIGATE call (line 24), then if segment ID of the considered brick is equal to searched segment ID (line 24), the segment can be validly added to the message (lines 25-26); otherwise, if the considered segment is not optional (line 27), then the segment cannot be added to the message (lines 28-29); in the other case, get next brick and repeat procedure (line 30)

line 31: return an ERROR if the brick type is not recognized

line 32: ID of the incoming segment could not be found in the message structure; therefore the segment cannot be validly added to the message being constructed

4.3.4 Next segment candidate

In the process of automatic generation the messages a completely different type of information is needed: possible segments which can be added to the message in the current moment of generation. This information is provided by the algorithm NEXT_SEGMENT_CANDIDATE, presented on Figure 4.4.

Figure 4.4 NEXT_SEGMENT_CANDIDATE algorithm

Input

brick — the first element (*message-brick*) of the proper message (with determined version, if exists) configuration; it can only be a segment, as all messages have to start with MSH segment

position — position of the last detected segment, or 0, when no positions where detected up to now.

> **Warning**
>
> For correct functioning of this algorithm, feedback of the calling routine is necessary for this parameter. Thus, first call to NEXT_SEGMENT_CANDIDATE should be with the position equal to 0 (zero), while next with the position equal to the one which was returned in the previous call to this function for a specified message or message version (if defined).

forbidden segments set — a set of segments which cannot be returned as a result of the algorithm

> **Warning**
>
> For correct functioning of this algorithm, feedback of the calling routine is necessary for this parameter. The calling routine should maintain a set of forbidden segments, i.e. segments which are not able to be generated in the current moment. As the result of this algorithm is the candidate segment ID, the calling routine should determine if it is possible to generate a segment of this ID. If it is possible, the calling routine should clear the forbidden segments set. If it is not possible, the returned candidate segment ID should be added to the set of forbidden segments, and NEXT_SEGMENT_CANDIDATE algorithm should be called again with the updated set.

Output

candidate segment ID — ID of the segment which is the valid candidate to generate in the particular moment

Stop condition

Algorithm stops in the case of:

- successfully determining ID of candidate segment (lines: 21-22, and 26-27)
- returning an ERROR value (lines: 2, 15, 30, and 31)
- navigating until the end of the message without finding the candidate segment; detection of this state is denoted below as: *MessageFinished()* condition (line: 4)

```
NEXT_SEGMENT_CANDIDATE(brick, position, forbidden_segments_set)

1   if NULL = brick
2   then return ERROR

// Phase 1: navigate to appropriate position
3   NAVIGATE(brick, position)

// Phase 2: find a position of a provided segment
4   while not FinishedScanning() and not MessageFinished()

5      if Type(brick) = GROUP_OF_SEGMENTS
6         TreatBrickAsGroup(brick)
7         brick := GetFirstBrickOfGroup(brick)
8         NEXT_SEGMENT_CANDIDATE(brick, position, forbidden_segments_set)
9         if not FinishedScanning() then
10           if GroupCanRepeat(brick) = false then return;
11           else if GroupWasScannedTwoTimes(brick) = false
12                   then GroupWasScannedTwoTimes(brick) := true
13                        ScanGroupAgain(brick)
14                else if GroupIsOptional(brick) = true then return
15                     else return ERROR

16     else if Type(brick) = SEGMENT
17        TreatBrickAsSegment(brick)
18        if FirstFoundSegment(brick) = true
19           if SegmentCanRepeat(brick) = true
20a             and SegmentIsForbidden(forbidden_segments_set, brick)
20b                 = false
21              then FinishedScanning()
22                   return SegmentID(brick)
23              else brick := GetNextBrick(brick)
24        else
25a          if SegmentIsForbidden(forbidden_segments_set, brick)
25b          = false
26           then FinishedScanning()
27                return SegmentID(brick)
28           else if SegmentIsOptional(brick) == true
29                then brick := GetNextBrick(brick)
30                else return ERROR

31     else return ERROR
```

NEXT_SEGMENT_CANDIDATE algorithm notes

lines 1-2: check if the brick is a valid object

line 3: navigate to current position in the message (see: Section 4.3.1);

> **Warning**
>
> This function is being called only once during the whole algorithm. In the recursive calls of NEXT_SEGMENT_CANDIDATE, call to NAVIGATE should be omitted.

line 4: main loop of the algorithm

lines 5-15: if the considered brick is a group of segments, it has to be treated as a group (line 6); starting with the first brick of the considered group (line 7) launch NEXT_SEGMENT_CANDIDATE algorithm recuresively (line 8), but starting from *phase 2* of the algorithm; after recursive call return, if not yet FinishedScanning() (line 9) check the following conditions:

- if group cannot repeat, finish the recursive call and return (line 10)
- otherwise, if group was not scanned to the end second time, scan the group again (lines 11-13)
- otherwise, if group is optional, finish the recursive call and return (line 14)
- else there is an error or in the strucutre, or in the forbidden segments set, or in the data used for generating the message — return ERROR (line 15)

lines 16-30: if the considered brick is a segment, it has to be treated as a segment (line 17); two cases are considered:

- if this segment is the segment which was found for the first time (line 18) after the call to NAVIGATE in line 3, then if segment is allowed to repeat (line 19) and is not forbidden (line 20a and b), then the segment is a valid candidate and can be returned (lines 21-22); otherwise, get next brick and repeat procedure (line 23)
- in the other case, if the the considered segment is not the first segment found after NAVIGATE call (line 24), then if it is not forbidden (lines 25a and b), then the segment is a valid candidate and can be returned (lines 26-27); else if the segment is optional (line 28), get next brick and repeat procedure (line 29); in the other case there is an error or in the strucutre, or in the forbidden segments set, or in the data used for generating the message — return ERROR (line 30)

line 31: return an ERROR if the brick type is not recognized

4.4 Database implementation limitations

Mapping presented in Section 3.5.3 give the possibility of storing the data of the field of the message in different storage column in different relation in (possibly different) database(s). Nevertheless, such possibility has to put some limitations on the database, or on the structure of the relation, as processing database queries has to be done in field-by-field fashion. These limitations are presented below.

4.4.1 Writting messages

HL7 Message Identifier column

As the database queries are being processed in field-by-field fashion, several fields of a message can be stored in the same relation. In order not to add each time a new tuple to a relation, a unique message identifier has to be provided, and called **HL7 Message Identifier**. Thus, all relation used for storing the data from interpreted messages have to define a column named: HL7_IG_MESSAGE_ID.

Caching DBRelations

Each time data is being written to the database, a proper *Database writer* should remember the relation and database in which it had written the data. This is called **DBRelation Cache**

Determining query type

Depending on if the DBRelation which suppose to be used while storing the data is in the DBRelation Cache, the proper query is being generated, in a manner shown on Figure 4.5.

Figure 4.5 *Storing data query type*

```
1   Determine the DBRelation which suppose to be used
2   If the DBRelation is in the DBRelation Cache then
3a     query := UPDATE :relation SET :column_name = :data_to_be_written
3b              WHERE HL7_IG_MESSAGE_ID = :hl7_ig_message_id
4   else
5      Put DBRelation into DBRelation Cache
6a     query := INSERT INTO :relation (:column_name, HL7_IG_MESSAGE_ID)
6b              VALUES(:data_to_be_written, :hl7_ig_message_id)
```

4.4.2 Reading messages

HL7 Message Identifier column

During the process of reading the data from database for automatic generation of the messages, the same type of *HL7 Message Identifier*, as defined in the section above, is used. It implies existance of the column HL7_IG_MESSAGE_ID in all the relations which store the data used for generating the messages.

field_name _HL7IGEN column

Nevertheless, as the reading data is also conducted in field-by-field fashion, each of the fields has to have an additional column, *data_column_name*_HL7IGEN, for determining if this field was already used or not for generating a message. The values of this field are: GENERATE — use this field for generating message data, and GENERATED — this field was already used for generating the message data.

ROWID-like column

Another limitation is that the relation used for generating messages data has to have a *ROWID-like* column. This column allows to generate repetitions of the same segment. It should be managed automatically by the database.

Sub-queries

In order to be able to generate repeated segments, the database should allow sub-queries in the WHERE clause.

Determining query type

Thus, the proper query is being generated in a manner shown on Figure 4.6.

> **Important**
>
> When reading the data from database for generating the message data, 2 queries have to be run: first, reading the data indeed, second, updating the appropriate field disallowing use of the same field in other repetition of the segment.

Figure 4.6 *Reading data query type*

```
SELECT query:
 1   control_column_name := column_name + "_HL7IGEN"
```

```
2   SELECT :column_name FROM :relation WHERE
3      HL7_IG_MESSAGE_ID = :hl7_ig_message_id AND
4      ROWID_LIKE_COLUMN =
5         ( SELECT MIN(ROWID_LIKE_COLUMN) WHERE
6              HL7_IG_MESSAGE_ID = :hl7_ig_message_id AND
7              control_column_name = "GENERATE"
8         )

UPDATE query:

1   control_column_name := column_name + "_HL7IGEN"
2   UPDATE :relation SET control_column_name = "GENERATED" WHERE
3      HL7_IG_MESSAGE_ID = :hl7_ig_message_id AND
4      ROWID_LIKE_COLUMN =
5         ( SELECT MIN(ROWID_LIKE_COLUMN) WHERE
6              HL7_IG_MESSAGE_ID = :hl7_ig_message_id AND
7              control_column_name = "GENERATE"
8         )
```

5 Conclusions

Building an interpreter and generator of the HL7 Standard messages is not an easy task. Even though the HL7 Standard itself is well defined, it allows many exceptions, and is not free of errors and inconsistencies. Nevertheless, some assumptions which have been taken allowed to provide a functional interpreter and generator of the HL7 Standard messages.

Presented *Messages configuration* allows to store information about all the messages defined in the HL7 Standard, with all the versions of the messages, and local variations. It is possible to define and add new segments and messages, as well as adapt existing messages to the specific needs of the healthcare organization. Apart from that, the *Database connectivity configuration* allows in a highly flexible manner to store all the data which are being exchanged with the use of the HL7 Standard messages protocol.

Moreover, the functionality of the HL7API library can be easily broaden. Addition of new databases being used and/or data sources/destinations for receiving/sending messages is as simple as implementing new classes serving a new type of database, source and/or destination. The User has direct access to all the existing classes, and is able to add new, correct exisitng, or even change the functionality of them.

Bibliography

[Borland Web page] *http://www.borland.com/us/*.

[Borland C++ Builder Web page] *http://www.borland.com/us/products/cbuilder/*.

[Chameleon] *http://www.interfaceware.com/chameleon.html*.

[Document Type Definition] *http://www.w3.org/XML/*.

[HL7 Standard specification] Larry Reis, Mark Shafarman, and Mark Tucker. *HL7 Standard specification*. HL7 Standard specification.

[HL7 Web page] *http://www.hl7.org/*.

[Iguana] *http://www.interfaceware.com/iguana.html*.

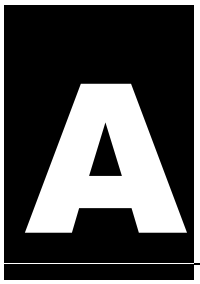
[OCI Programmer's Guide] *http://lbd.epfl.ch/f/teaching/courses/oracle8i/server.815/a67846/toc.htm*.

[OCI Web page] *http://www.oracle.com/technology/tech/oci/*.

[Oracle Web page] *http://www.oracle.com/*.

[Scan7] *http://www.7scan.com/*.

[TinyXml Web page] *http://sourceforge.net/projects/tinyxml/*.



Thesis CD contents

The encloded CD-ROM contains:

- /bin — HL7API library compiled in DEBUG and RELEASE mode
- /doc — usage instruction (in polish) and the API documentation
- /examples — examples of use of the HL7API library
- /hl7api-conf — all configuration files described in the thesis
- /src — sources of the HL7API library
- /thesis — thesis in Adobe PDF format and its sources

B Configuration examples

This appendix contains several configuration examples.

B.1 Table

Figure B.1 presents an example of empty, user-defined table configuration file.

Figure B.1 *Empty user-defined table example*

```
<?xml version="1.0" standalone="no" ?>
<TABLE NUMBER="44" TYPE="User" NAME="Contract code" />
```

Figure B.2 presents an example of non-empty, Standard-defined table configuration file.

Figure B.2 *Non-empty Standard-defined table example*

```
<?xml version="1.0" standalone="no" ?>
<TABLE NUMBER="27" TYPE="HL7" NAME="Priority">
   <ITEM VALUE="S" DESCRIPTION="Stat (do immediately)" />
   <ITEM VALUE="A" DESCRIPTION="As soon as possible (a priority lower
                                than stat)" />
   <ITEM VALUE="R" DESCRIPTION="Routine" />
   <ITEM VALUE="P" DESCRIPTION="Preoperative (to be done prior to
                                surgery)" />
   <ITEM VALUE="T" DESCRIPTION="Timing criticial (do as near as
                                possible to requested time)" />
</TABLE>
```

B.2 Table index

Figure B.3 presents an example of a part of table index configuration file.

Figure B.3 *Part of a table_index.xml example*

```
<?xml version="1.0" standalone="no"?>
<TABLE_INDEX>
   <TABLE NUMBER="1" FILE="tab_0001.xml"/>
   <TABLE NUMBER="2" FILE="tab_0002.xml"/>
   <TABLE NUMBER="3" FILE="tab_0003.xml"/>
   <TABLE NUMBER="4" FILE="tab_0004.xml"/>
   <TABLE NUMBER="5" FILE="tab_0005.xml"/>
   (...)
   <TABLE NUMBER="4000" FILE="tab_4000.xml"/>
</TABLE_INDEX>
```

B.3 Fields

Figure B.4 presents an example of a part of fields configuration file.

Figure B.4 Part of a fields.xml example

```
<?xml version="1.0" standalone="no"?>
<FIELDS>
   <FIELD_DATA SEQ="01" NAME="&amp;quot;Authorizing Payor, Plan
      Code&amp;quot;" ITEM="1146" SEG="AUT" CHP="11" LEN="200" DT="CE"
      REP="No" TABLE="72,,,72" REQ="O" TYPE="HL7"/>
   <FIELD_DATA SEQ="01" NAME="Accident Date/Time" ITEM="527" SEG="ACC"
      CHP="6" LEN="26" DT="TS" REP="No" REQ="O" TYPE="HL7"/>
   (...)
   <FIELD_DATA SEQ="01" NAME="Error Code and Location" ITEM="24"
      SEG="ERR" CHP="2" LEN="80" DT="CM" REP="Yes" TABLE=",,,357"
      REQ="R" TYPE="HL7">
      <CM_COMPONENT SEQ="01" TYPE="ST" NAME="segment ID" REQ="R"/>
      <CM_COMPONENT SEQ="02" TYPE="NM" NAME="sequence" REQ="R"/>
      <CM_COMPONENT SEQ="03" TYPE="NM" NAME="field position" REQ="R"/>
      <CM_COMPONENT SEQ="04" TYPE="CE" NAME="code identifying error"
         REQ="R"/>
   </FIELD_DATA>
   (...)
   <FIELD_DATA SEQ="06" NAME="Reference (Normal) Range - Ordinal &amp;amp;
      Continuous Obs" ITEM="631" SEG="OM2" CHP="8" LEN="200" DT="CM"
      REP="No" REQ="O" TYPE="HL7">
      <CM_COMPONENT SEQ="01" TYPE="CM" NAME="reference (normal) range"
         REQ="R">
         <CM_SUB_COMPONENT SEQ="01" TYPE="ST" NAME="low value"
            REQ="R"/>
         <CM_SUB_COMPONENT SEQ="02" TYPE="ST" NAME="high value"
            REQ="R"/>
      </CM_COMPONENT>
      <CM_COMPONENT SEQ="02" TYPE="IS" NAME="sex" REQ="R"/>
      <CM_COMPONENT SEQ="03" TYPE="CM" NAME="age range" REQ="R">
         <CM_SUB_COMPONENT SEQ="01" TYPE="ST" NAME="low value"
            REQ="R"/>
         <CM_SUB_COMPONENT SEQ="02" TYPE="ST" NAME="high value"
            REQ="R"/>
      </CM_COMPONENT>
   </FIELD_DATA>
   (...)
</FIELDS>
```

B.4 Segments

Figure B.5 presents an example of a part of segments configuration file.

Figure B.5 Part of a segments.xml example

```
<?xml version="1.0" standalone="no"?>
<SEGMENTS>
   <SEGMENT_DATA SEGMENT="ACC" DESCRIPTION="Accident segment"
      CHAPTER="6" TYPE="HL7" />
   <SEGMENT_DATA SEGMENT="ADD" DESCRIPTION="Addendum segment"
      CHAPTER="2" TYPE="HL7" />
   <SEGMENT_DATA SEGMENT="AIG" DESCRIPTION="Appointment information
      - general resource" CHAPTER="10" TYPE="HL7" />
   <SEGMENT_DATA SEGMENT="AIL" DESCRIPTION="Appointment information -
      location resource segment" CHAPTER="10" TYPE="HL7" />
   <SEGMENT_DATA SEGMENT="AIP" DESCRIPTION="Appointment information -
      personnel resource segment" CHAPTER="10" TYPE="HL7" />
```

```
   <SEGMENT_DATA SEGMENT="AIS" DESCRIPTION="Appointment information -
      service segment" CHAPTER="10" TYPE="HL7" />
(...)
</SEGMENTS>
```

B.5 Single-version message

Figure B.6 presents an example of a part of messages configuration file containing single-version message.

Figure B.6 *Single-version message example*

```
<MESSAGE_DATA MESSAGE="VXX" DESCRIPTION="Vaccination query response
   with multiple PID matches" CHAPTER="4" TYPE="HL7">
   <SEGMENTS_DEF>
      <SEGMENT SEQ="01" SEG="MSH" OPTIONAL="No" REPEAT="No"/>
      <SEGMENT SEQ="02" SEG="MSA" OPTIONAL="No" REPEAT="No"/>
      <SEGMENT SEQ="03" SEG="QRD" OPTIONAL="No" REPEAT="No"/>
      <SEGMENT SEQ="04" SEG="QRF" OPTIONAL="Yes" REPEAT="No"/>
      <GROUP_OF_SEGMENTS NAME="Group1" OPTIONAL="No" REPEAT="Yes">
         <SEGMENTS_DEF>
            <SEGMENT SEQ="05" SEG="PID" OPTIONAL="No" REPEAT="No"/>
            <SEGMENT SEQ="06" SEG="NK1" OPTIONAL="Yes" REPEAT="Yes"/>
         </SEGMENTS_DEF>
      </GROUP_OF_SEGMENTS>
   </SEGMENTS_DEF>
</MESSAGE_DATA>
```

B.6 Multi-version message

Figure B.7 presents an example of a part of messages configuration file containing multi-version message.

Figure B.7 *Multi-version message example*

```
<MESSAGE_DATA MESSAGE="DSR" DESCRIPTION="Display response" CHAPTER="2"
   TYPE="HL7">
   <COMMON_START_SEGMENTS>
      <SEGMENTS_DEF>
         <SEGMENT SEQ="01" SEG="MSH" OPTIONAL="No" REPEAT="No"/>
      </SEGMENTS_DEF>
   </COMMON_START_SEGMENTS>
   <VERSION_SEGMENTS>
      <VERSION NAME="DSR_Q01" DESCRIPTION="DSR/Q01 - QRY/DSR -
         original mode display query - immediate response (event Q01)"
         DEFAULT="Yes">
         <CONDITION>
            <CND_DEF CND_SEQ="01" SEG_ID="MSH" FIELD_SEQUENCE="09"
               FIELD_COMPONENT_NUMBER="2" OPERAND="EQUAL"
               VALUE="Q01"/>
         </CONDITION>
         <SEGMENTS_DEF>
            <SEGMENT SEQ="02" SEG="MSA" OPTIONAL="No" REPEAT="No"/>
            <SEGMENT SEQ="03" SEG="ERR" OPTIONAL="Yes" REPEAT="No"/>
            <SEGMENT SEQ="04" SEG="QAK" OPTIONAL="Yes" REPEAT="No"/>
            <SEGMENT SEQ="05" SEG="QRD" OPTIONAL="No" REPEAT="No"/>
            <SEGMENT SEQ="06" SEG="QRF" OPTIONAL="Yes" REPEAT="No"/>
            <SEGMENT SEQ="07" SEG="DSP" OPTIONAL="No" REPEAT="Yes"/>
            <SEGMENT SEQ="08" SEG="DSC" OPTIONAL="Yes" REPEAT="No"/>
```

```
            </SEGMENTS_DEF>
         </VERSION>
         <VERSION NAME="DSR_Q03" DESCRIPTION="DSR/Q03 - DSR/ACK -
            deferred response to a query (event Q03)">
            <CONDITION>
               <CND_DEF CND_SEQ="01" SEG_ID="MSH" FIELD_SEQUENCE="09"
                  FIELD_COMPONENT_NUMBER="2" OPERAND="EQUAL"
                  VALUE="Q03"/>
            </CONDITION>
            <SEGMENTS_DEF>
               <SEGMENT SEQ="02" SEG="MSA" OPTIONAL="Yes" REPEAT="No"/>
               <SEGMENT SEQ="03" SEG="QRD" OPTIONAL="No" REPEAT="No"/>
               <SEGMENT SEQ="04" SEG="QRF" OPTIONAL="Yes" REPEAT="No"/>
               <SEGMENT SEQ="05" SEG="DSC" OPTIONAL="Yes" REPEAT="No"/>
            </SEGMENTS_DEF>
      </VERSION>
   </VERSION_SEGMENTS>
</MESSAGE_DATA>
```

B.7 Databases

Figure B.8 presents an example of a part of databases configuration file.

Figure B.8 *Part of a databases.xml example*

```
<?xml version="1.0" standalone="no"?>
<DATABASES>
   <DATABASE ID="GOO" TYPE="OCI" DESCRIPTION="Test database for
         developping in OCI connection">
      <DB_OCI USER="goo" PASS="123"/>
   </DATABASE>
   <DATABASE ID="GOOMYSQL" TYPE="ODBC" DESCRIPTION="Test database for
         developping in ODBC connection">
      <DB_ODBC DATA_SOURCE_NAME="HL7APIDB"/>
   </DATABASE>
</DATABASES>
```

B.8 Relations

Figure B.9 presents an example of a part of relations configuration file.

Figure B.9 *Part of a relations.xml example*

```
<?xml version="1.0" standalone="no"?>
<RELATIONS>
   <RELATION ID="R_ACK" DATABASE="GOO" NAME="ACK_MESSAGES"
      DESCRIPTION="Test relation for ACK messages"/>
   <RELATION ID="R_ACK_01" DATABASE="GOO"
      NAME="ACK_MESSAGES_SOME_DATA"
      DESCRIPTION="Test relation for ACK messages number 2"/>
</RELATIONS>
```

B.9 Mappings

Figure B.10 presents an example of a part of mappings configuration file.

Figure B.10 Part of a mappings.xml example

```
<?xml version="1.0" standalone="no"?>
<MAPPINGS>
   <MAPPING MESSAGE="ACK" SEGMENT="MSH" SEGMENT_POSITION="01"
         DEFAULT_DBRELATION="R_ACK"/>
   <MAPPING MESSAGE="ACK" SEGMENT="MSA" SEGMENT_POSITION="01"
         DEFAULT_DBRELATION="R_ACK">
      <MAPPING_ITEM DBRELATION="R_ACK_01" ITEM="18" COLUMN="MSA_01"/>
      <MAPPING_ITEM DBRELATION="R_ACK_01" ITEM="10" COLUMN="MSA_02"/>
      <MAPPING_ITEM ITEM="20" COLUMN="MSA_03"/>
      <MAPPING_ITEM ITEM="21" COLUMN="MSA_04"/>
      <MAPPING_ITEM ITEM="22" COLUMN="MSA_05"/>
      <MAPPING_ITEM ITEM="23" COLUMN="MSA_06"/>
   </MAPPING>
</MAPPINGS>
```

B.10 Safe-storage

Figure B.11 presents an example of a part of safe-storage configuration file.

Figure B.11 Part of a safe-storage-relation.xml example

```
<?xml version="1.0" standalone="no"?>
<SAFE_STORAGE_RELATION>
   <SSRELATION DATABASE="GOO" NAME="SAFE_STORAGE" COLUMN="MSGS"
      DESCRIPTION="Test relation for safe storage"/>
</SAFE_STORAGE_RELATION>
```