EDI Claims Debatcher

Algorithm (Notes)

Paul Short

Contents

[1 Purpose 0](#_Toc11317806)

[2 Algorithm 0](#_Toc11317807)

[2.1 Interface 1](#_Toc11317808)

[2.2 Runtime Behavior 1](#_Toc11317809)

[2.2.1 State 1](#_Toc11317810)

[2.2.2 Segment Type 1](#_Toc11317811)

[2.2.3 HLState Type 2](#_Toc11317812)

[2.2.4 Details (in Psuedocode) 2](#_Toc11317813)

[3 Handling Details 4](#_Toc11317814)

[3.1 Handling Interchange Control, Function Group and Transaction Set (ISA, GS and ST) envelope: 4](#_Toc11317815)

[3.2 Handling the Submitter and Receiver Names (Loop 1000A and 1000B) 4](#_Toc11317816)

[3.3 Handling Hierarchical levels (HL) 4](#_Toc11317817)

[3.3.1 Billing Provider Hierarchical Level (2000A) 5](#_Toc11317818)

[3.3.2 Subscriber Hierarchical Level (2000B) 5](#_Toc11317819)

[3.3.3 Patient Hierarchical Level (2000C) 5](#_Toc11317820)

# Purpose

The EDI Debatcher is a file splitting algorithm used to split large edi files into smaller more manageable chunks for processing. Usually the parent edi files received are of large sizes some of them estimated to be in the order of Gigabytes. The debatcher will read in the file and split them depending on the segments inside of it right up to the CLM (Claim) level and write the files out for further processing.

# Algorithm

## Interface

Our algorithm takes in batched EDI claims and breaks them down to smaller, stand-alone EDI claims. The input to the algorithm is an input stream of strings; the output is an output stream of strings.

## Runtime Behavior

Nested execution of the algorithm is stacked like this (from outer to inner):

**Read-Interchange-Controls**(InputStream)

**Read-Function-Groups**(InputStream)  
 **Read-Transaction-Sets**(InputStream)  
 **Read-Header**(InputStream)  
 **Read-Hierarchical-Levels**(InputStream)

**Read-Claims**(InputStream)

**Read-Claim**(InputStream)

**Write-Claim**(Read-\* Buffers, OutputStream)

### State

|  |  |  |
| --- | --- | --- |
| **Variable** | **Type** | **Description** |
| ISA Buffer | String | Block of text between ISA\* and GS\* (Interchange Control Envelope). |
| ISA Id | String | Interchange Control Envelope Identifier (ISA13). |
| Field Delimiter | char | Character at ISA position 104. |
| Segment Delimiter | char | Character at ISA position 106 (last char in ISA). |
| Current Segment | **Segment** | Temp variable to hold current segment as we advance. |
| GS Buffer | String | Block of text between GS\* and ST\* (Function Group Envelope). |
| GS Id | String | Function Group Envelope Identifier. |
| Header Buffer | String | Block of text between ST\* and first HL\*: includes ST, BHT, Submitter Name Loop (1000a) and Receiver Name Loop (1000b). |
| ST Counter | int | Count of all segments within an ST-SE block. |
| Hierarchical Loop Stack | **Stack<HLState>** | Stack typically contains two instances of the HLState type: Billing Provider and Subscriber, but may contain three (Patient). |
| Hierarchical Loop Counter | int | ST scope. Starts at 1, incremented by 1 each time an HL is used in the transaction set. Resets to 1 on the next transaction set. Passed on as the HL Parent Id when an HL is created. |
| CLM Buffer | String | Block of text between CLM\* and CLM\*|HL\*|SE\*. |
| CLM Counter | int | ST scope. Sequential claim id (one-up counter). |

### Segment Type

|  |  |  |
| --- | --- | --- |
| **Variable** | **Type** | **Description** |
| Id | String | Type of current segment (ISA, CLM, HL, etc.) Typically, a 2-3 char id. |
| Buffer | String | Block of text to hold segment data |

### HLState Type

|  |  |  |
| --- | --- | --- |
| **Variable** | **Type** | **Description** |
| HL Buffer | String | Block of text between HL\* and HL\*|CLM\* |
| HL Id | int | See *Hierarchical Loop Counter* |
| HL Parent Id | Int | N/A (0) for top-most HL (Billing Provider). Subscriber/Patient would use Id of respective Billing Provider/Subscriber parent. |
| HL Level Code | Int | 20=Billing provider; 22=Subscriber; 23=Patient |
| HL Child Code | Int | 0=no subordinate HL; 1=has subordinate HL |

### Details (in Psuedocode)

The general algorithm definition (assuming certain stream and error handling behaviors):

**Debatch (InputStream, OutputStream)**  
1 while not InputStream.End  
2 do Read-Interchange-Controls(InputStream)

**Read-Interchange-Controls(InputSteam)**  
1 Consume-Whitespace(InputStream)  
2 if InputStream.End then return  
3 [ISA Buffer] = InputStream.Read-Advance(106)  
4 [ISA Id] = [ISA Buffer](91, 9)  
5 [Field Delimiter] = [ISA Buffer](104, 1)  
6 [Segment Delimiter] = [ISA Buffer](106, 1)  
7 Read-Function-Groups(InputStream)  
// TBD put in Loop

**Consume-Whitespace(InputStream)**  
1 while not InputStream.End and InputStream.current-char is whitespace  
2 do InputStream.Next-Char

**Read-Function-Groups(InputStream)**1 [Current Segment] = Next-Segment(InputStream)  
2 if [Current Segment.Id] is not GS then Exit-Error  
3 [GS Buffer] = [Current Segment.Buffer]  
4 [GS id] = Read-Field([GS Buffer], 6)  
5 Read-Transaction-Sets(InputStream)  
// TBD put in Loop

**Next-Segment(InputStream)**  
1 while not InputStream.End and InputStream.Current-char is not [Segment Delimiter]  
2 do InputStream.Next-Char  
3 [ST Counter] = [ST Counter] + 1  
4 id = InputStream.Seek([Field Delimiter])  
5 return id

**Exit-Error***stop parsing, exit as exception/report error*

**Read-Field(Segment, N)**  
1 *// Several ways to do this; here we seek the field rather than store/index it*  
2 n = -1 *// when n is 0 we get segment id, when n is 1 we get field 1*3 buffer = empty  
4 for char c in Segment.Buffer  
5 if c is [Field Delimiter] then   
6 n = n + 1  
7 if n = N + 1 then break for  
8 if n = N then buffer.append(c)   
9 return buffer

**Read-Transaction-Set**s(InputStream)  
 while [Current Segment.Id] is not ST *// not ST => GE*  
 [ST Counter] = 0  
 [Current Segment] = Next-Segment(InputStream)  
 if [Current Segment.Id] is not ST then Exit-Error  
 control number = Read-Field([Current Segment], 2) *// ST02*  
 doRead-Header(InputStream)  
 [Current Segment] = Next-Segment(InputStream)  
 if [Current Segment.Id] is not SE then Exit-Error  
 if control number is not Read-Field([Current Segment], 2) then Exit-Error *// SE02* [Current Segment] = Next-Segment(InputStream)  
  
// TBD Finish putting in Loop  
// [ST Counter] logic probably isn’t right—we should count what we are going to write, which isn’t exactly the same as what we read  
  
**Read-Header**(InputStream)  
 // TBD  
Read-Hierarchical-Levels(InputStream)

**Read-Hierarchical-Levels**(InputStream)  
 // TBD  
Read-Claims(InputStream)  
  
**Read-Claims**(InputStream)  
 // TBD  
Read-Claim(InputStream)

**Read-Claim**(InputStream)  
 // TBD

**Write-Claim**(Read-\* Buffers, OutputStream)  
 // TBD

# Handling Details

## Handling Interchange Control, Function Group and Transaction Set (ISA, GS and ST) envelope:

The ISA, GS, ST can be read in as strings, since they will be constant for the entire set (ISA to IEA)

1. Read the stream one character at a time; and check for whitespace (eat up the whitespace). Keep advancing until the first actual character is reached.
2. Initialize a String buffer and read the characters and add to buffer until either delimiter is reached( Segment, Element)
3. Check the buffer for the values
4. If the characters are “ISA”; then read the first 102 bytes (Actual value from ISA to the delimiter is 106, but we consume the first 4 bytes which is ‘I’,’S’,’A’,’\*’), so 5 to 102 will contain the delimiters needed.
5. Set the delimiters and read the ISA into a string.
6. Advance the character from the 106th character onwards to read the GS block.
7. Perform steps 1) to 3)
8. If the buffer contains “GS” then read the fields from ‘\*’ onwards till the segment delimiter(generally tilde ~)
9. Read into GS string object
10. Advance the character from the last segment delimiter encounter
11. Perform steps 1) to 3)
12. If the buffer contains “ST”, read the fields from ‘\*’ to the segment delimiter(~)

## Handling the Submitter and Receiver Names (Loop 1000A and 1000B)

1. Repeat the same for NM1, PER and NM1 same as ISA, GS and ST

## Handling Hierarchical levels (HL)

The algorithm makes use of the HL segments in the transaction to make a determination if there are any further HL segments to start splitting the sections.

There can be at most 2 HL sections one for Billing Provider (2000A) HL and one for Subscriber (2000B) HL. There may or may not be any Patient (2000C) HL.

HL\*{ID Number}\*{Parent ID Number}\*{Level Code}\*{Child Code} ~

NOTE: Support for Patient loops is implemented for completeness with the EDI specification and potential reuse outside of the EDCP project. It is not needed for EDCP. Patient loops would only be used in the case of reporting a dependent. Since neither Medicare/Medicaid members have dependents (everyone is a subscriber), this loop should not be submitted on any of the encounter data that we receive for the EDCP project.

We would need to create a STACK of type Hierarchical Level which would contain the HL ID Number, HL Parent ID Number, HL Level Code and a Child Code.

1. When we scan and get the HL segment; read the elements in the segment up to the HL01;
2. Read the count and maintain it as a running counter; first HL will be marked as 1 and any subsequent HL with be incremented by 1.
3. Update the stack with the ID number. Check for HL02 this will be blank for first HL(Billing Provider), maintain the counter and update with 0, this is the parent counter indicating that the first HL doesn’t have any parents.
4. Read the next element HL03, and save the value ; this will be the Level code id; first one will be 20 for BP
5. Read the next element HL04 and save it as child code. This will indicate if there are any further child HL segments. First one will have value of 1.
6. Check the next segment for HL for Subscriber HL; save the ID number, parent ID, level code, check for child code; if child code = 1 there would be another HL else if 0 then the Subscriber has the claim loop 2300 inside it.
7. Advance to read the loop2300
8. Read the CLM to the next HL segment
9. Repeat from 1)

### Billing Provider Hierarchical Level (2000A)

HL\*{ID Number}\*{Parent ID Number}\*{Level Code}\*{Child Code} ~

ID Number = 1

Indicating this is the first occurrence of the HL, there no HL above this

Parent ID Number = <blank>

Indicating there are no parents for this HL

Level Code = 20

Information source (unique ID for BP HL=20)

Child Code = 1

Indicates there can be subscriber HL (2000B) as a subordinate. Every BP HL(2000A) must have atleast one child HL loop to follow.

e.g. HL\*1\*\*20\*1~

### Subscriber Hierarchical Level (2000B)

HL\*{ID Number}\*{Parent ID Number}\*{Level Code}\*{Child Code} ~

ID Number = 2

Indicating this is the second occurrence of the HL, first being BP HL

Parent ID Number = 1

Indicating that the Parent ID for this HL is 1; which is the Billing Provider HL

Level Code = 22

Subscriber HL.

Child Code = 0 or 1

Indicates there can be patient HL (2000C) as a subordinate if 1 or 0 if there are no subordinates.

e.g. HL\*2\*1\*22\*1~

### Patient Hierarchical Level (2000C)

HL\*{ID Number}\*{Parent ID Number}\*{Level Code}\*{Child Code} ~

ID Number = 3

Indicating this is the third occurrence of the HL, first two being BP HL and Subscriber HL

Parent ID Number = 2

Indicating that the Parent ID for this HL is 2; which is the Subscriber HL

Level Code = 23

Patient HL; the subscriber and patient are not the same.

Child Code = 0

No subordinate HL segment in the current transaction (ST-SE)

e.g HL\*3\*2\*23\*0~