Annex 1. The package.

a) Class *EventLogLoader*

```
"""
Author: Daria Alexander

The class EventLogLoader loads a JSON file into Pandas dataframe. It also adds new columns to
the dataframe. Those columns will help to calculate fluency measures in the class
MessageHandling
"""

import pandas as pd
import json
import numpy as nm

class eventLogLoader():

    def __init__(self):
        self.events = None # Pandas dataframe

    def load(self, path_json):
        """
        loads a JSON file into a Pandas Dataframe
        """
        try :
            with open(path_json, encoding='utf-8') as data_file:
                data = json.loads(data_file.read())
                self.events = pd.DataFrame(data)
        except (FileNotFoundError, IOError):
            print ("File not found")

        # change the format of the column event_time to datetime64
        self.events['event_time'] = pd.to_datetime(self.events['event_time'])


    def extract_length_diff(self):
        """
        extracts the length difference between the previous line and the current line
        """
        dfr = self.events[self.events.event_type_t == 'Text change']
        self.events['characters_delta'] =\
         dfr.event_text.str.len() - dfr.shift(1).event_text.str.len()
        self.events.loc[self.events.event_type_t == 'TTS voicing start', 'characters_delta'] = 1



    def extract_pauses(self, minimalPauseTreshold = 300):
        """
        :param minimal pause threshold: 250, 300, 500, 1000, 2000
        outputs True if the pause > then a chosen threshold, False otherwise
        """
        self.events = self.events.assign(timeDiff = self.events.event_time.diff())
        self.events['pauses'] = self.events.timeDiff\
        > pd.Timedelta(str(minimalPauseTreshold)+'ms')
        #self.events['Pause500'] = self.events.timeDiff > pd.Timedelta('250ms')
        #self.events['Pause500'] = self.events.timeDiff > pd.Timedelta('300ms')
        #self.events['Pause500'] = self.events.timeDiff > pd.Timedelta('500ms')
        #self.events['Pause1000'] = self.events.timeDiff > pd.Timedelta('1000ms')
        #self.events['Pause2000'] = self.events.timeDiff > pd.Timedelta('2000ms')

    def extract_revisions(self):
        """
        outputs True if the difference between the previous and the current line <= - 1
        False otherwise
        """
        self.events['revisions'] = self.events.characters_delta <= -1

    def save_data(self):
        """
        saves the data in the test file
```

```
        """
        writer = self.events.to_csv("data.csv", sep=';')

    def get_events(self):
        """
        gets the dataframe events
        """
        return self.events


# we are doing the main
if __name__ == '__main__' :

    matches = eventLogLoader()
    matches.load('logs_events.json')
    matches.extract_length_diff()
    matches.extract_pauses()
    matches.extract_revisions()
    matches.save_data()
    print(matches.events.iloc[1])
    print(matches.events.iloc[1])
    res = matches.get_events()
    print(res)
```

b) Class MessageHandling

```
"""
Author: Daria Alexander

This class computes 16 fluency measures and also basic variables that are necessary
for the calculations of those measures. The results are stored in the dictionary result.
"""

import pandas as pd
import eventLogLoader as logload
import datetime as d_t
from datetime import datetime
import csv
import numpy as nm
from nltk.tokenize import RegexpTokenizer

class messageHandling():

    def __init__(self, contents, number):
        self.contents = contents # Pandas dataframe
        self.messagenumber = number # Message number
        self.previousmessage = None # initialising the previous message
        self.result = {} # the dictionary where the results will be stored

    def printmessage(self):
        print(self.messagenumber)
        print(self.contents)

    """
    basic variables

    we compute basic variables that will help us to calculate fluency measures

    TIME
    """



    def user_message_submit_time(self):
        """
        formula: last event of event type "End edit"
        :return the time when the user sends the message (Timestamp)
        """
        d = self.contents.event_time[self.contents.event_type_t == 'End edit'].iloc[-1]
        self.result['submit_time'] = d
        return self.result['submit_time']
```

```
def first_typing_event(self):
    """
    formula: first event of the type "Text change"
    :return: the first typing event (Timestamp)
    """
    l = list(self.contents.event_time[(self.contents.event_type_t \
    == 'Text change')].astype(datetime))
    ret = l[0]
     self.result['first_event'] = ret
    return self.result['first_event'] # we write the result in the dictionary


def system_message_reception_time(self):
    """
    formula: the event of the type "System message"
    :return: the moment when the system gets the message from the user(Timestamp)
    """
    d = self.contents.event_time[self.contents.event_type_t == 'System message']
    d = d.astype(datetime).values[0]
    self.result['message_reception_time'] = d
    return self.result['message_reception_time']


def user_message_duration(self):
    """
    formula: submit time - first typing event
    :return: the duration of user's message (Timedelta)
    """
    b = self.result['first_event']
    d = self.result['submit_time'] - b
    self.result['us_message_duration'] = d
    return self.result['us_message_duration']


def user_turn_pause_duration(self):
    """
    formula: first typing event of the current message - message reception time
     of the previous message
    :return: the duration of the turn pause(Timedelta)
    """
    d = d_t.timedelta(microseconds=0) # a module datetime import datetime
    a = self.result['first_event'] # it is the first event of the current message
    if self.previousmessage != None:
        b = self.previousmessage.result['message_reception_time'] #it is a reception
             #time of the previous message
        d = a -b # we extract the previous from the current
    self.result['us_turn_pause_duration'] = d
    return self.result['us_turn_pause_duration']


def user_turn_duration(self):
    """
    formula: turn pause duration + user message duration
    :return: user turn duration (Timedelta)
    """
    a = self.result['us_message_duration']
    b = self.result['us_turn_pause_duration']
    d = a + b
    self.result['us_turn_duration'] = d
    return self.result['us_turn_duration']



def pauses_duration(self):
    """
    Formula: sum (single pauses' duration)
    nb: we don't take the turn pause and the last event of the type "Text change"
     into account
    :return: the duration of the pauses (Timedelta)
    """
    r = 0
```

```
        self.result['pauses_duration'] = d_t.timedelta(microseconds=0)
        dt = self.contents[(self.contents['event_type_t'] == 'Text change')].iloc[1:-1]
         # iloc - we do not take the first element because it is a turn pause and we do not
         #take the last one because it comes after submit.
        a = dt[(dt['pauses'] == True)].sum()["timeDiff"] # if the pause is true,
         #we make a sum of all the pauses
        if a != 0:
            self.result['pauses_duration'] += a
        pe = self.contents.pauses[(self.contents['event_type_t'] == 'End edit')].iloc[-1]
        # we also take the final pause before submit(Timedelta)
        if pe == True:
            ep = self.contents.timeDiff[(self.contents['event_type_t'] == 'End edit')].iloc[-1]
            self.result['pauses_duration'] += ep
        return self.result['pauses_duration']


    def extract_p_bursts(self):
        """
        formula: user message duration - pauses duration
        :return: the duration of p-bursts(Timedelta)
        """
        self.result['p_bursts'] = d_t.timedelta(microseconds=0)
        md = self.result['us_message_duration']
        pd = self.result['pauses_duration']
        if md != 0:
            pb = md - pd
        if md != 0:
            self.result['p_bursts'] = pb
        return self.result['p_bursts']

    def revisions_duration(self):
        """
        formula: sum (single revisions' duration)
        nb: we do not take the first event of type "Text change" because
        a revision cannot be performed on 0 characters
        :return: duration of revisions (Timedelta)
        """
        self.result['revisions_duration'] = d_t.timedelta(microseconds=0)
        dt = self.contents[(self.contents['event_type_t'] == 'Text change')].iloc[1:]
         # we do not take the first event of type "Text change" because
         #there can be no revisions at that moment
        a = dt[(dt['revisions'] == True)].sum()["timeDiff"] # sum of revisions
        if a != 0:
            self.result['revisions_duration'] +=a
        dt1 = self.contents[(self.contents['event_type_t'] == 'End edit')] # we also take
         #the revision before submit(Timedelta)
        b = dt1[(dt1['revisions'] == True)].sum()["timeDiff"]
        if b != 0:
            self.result['revisions_duration'] += b #if the revision before "end edit" exists,
          #we add it to the duration of revisions
        return self.result['revisions_duration']

    def extract_r_bursts(self):
        """
        formula: user message duration - revisions duration
        :return: the duration of r-bursts (Timedelta)
        """
        self.result['r_bursts'] = d_t.timedelta(microseconds=0)
        md = self.result['us_message_duration']
        rd = self.result['revisions_duration']
        if md != 0:
            rb = md - rd
        if rb != 0:
            self.result['r_bursts'] = rb
        return self.result['r_bursts']

    """
    basic variables

    CHARACTERS
    """
```

```
def number_of_total_characters(self):
    """
    formula: sum (charactersDelta >= 1)
    :return: the total number of characters(int)
    """

    self.result['nb_of_tot_char'] = 0
    count = 0
    for i in self.contents['characters_delta']:
        if i >= 1:
            count +=i
    if count != 0:
        self.result['nb_of_tot_char'] +=count
    return self.result['nb_of_tot_char']

def number_of_deleted_characters(self):
    """
    formula = sum (charactersDelta <= -1)
    :return: the number of deleted characters(int)
    """
    self.result['nb_of_del_char'] = 0
    count = 0
    for i in self.contents['characters_delta']:
        if i <= -1:
            count +=abs(i)
    if count != 0:
        self.result['nb_of_del_char'] +=count
    return self.result['nb_of_del_char']

def number_of_final_characters(self):
    """
    formula: total characters - deleted characters
    :return: the number of final characters(int)
    """
    self.result['nb_of_fin_char'] = 0
    nb = self.result['nb_of_tot_char']
    nb2 = self.result['nb_of_del_char']
    res = nb-nb2
    if res != 0:
        self.result['nb_of_fin_char'] += res
    return self.result['nb_of_fin_char']

def number_of_revisions(self):
    """
    formula: sum (single revisions)
    :returns the number of revisions(int)
    """
    self.result['nb_of_revisions'] = 0
    for i in self.contents['revisions']:
        if i == True:
            self.result['nb_of_revisions']+=1
    dt1 = self.contents[(self.contents['event_type_t'] == 'End edit')] # we also take
    # the revision before submit(Timedelta)
    v = dt1['revisions'].values[-1]
    if v == True:
        self.result['nb_of_revisions']+=1
    return self.result['nb_of_revisions']


def number_of_pauses(self):
    """
    formula: sum (single pauses)
    :return: the number of pauses(int). Not including the first pause (turn pause)
     #and the last(submit) pause
    """
    self.result['nb_of_pauses'] = 0
    count = 0
    nb = self.contents[(self.contents['event_type_t'] == 'Text change')].iloc[1:-1]
     # we do not take the first pause into account because it is a turn pause.
    # We do not take the last event of type "text change" because it is not actual typing
    for i in nb['pauses']:
        if i == True:
```

```
            count +=1
    dt1 = self.contents[(self.contents['event_type_t'] == 'End edit')]
     # we also take the final pause before submit(Timedelta)
    v = dt1['pauses'].values[-1]
    if v == True: # if it is a pause, we add + 1 to pauses
        count += 1
    self.result['nb_of_pauses'] = count
    return self.result['nb_of_pauses']

def number_p_bursts(self):
    """
    formula: If the last event is pause then number of p-bursts == number of pauses.
    else number of p-bursts == number of pauses + 1
    :return: the number of p-bursts(int).
    """
    self.result['nb_of_p_bursts'] = 0
    dt1 = self.contents[(self.contents['event_type_t'] == 'End edit')]
    v = dt1['pauses'].values[-1] # if the last event (event of type "End edit") is a pause
    if v == True:
        self.result['nb_of_p_bursts'] = self.result['nb_of_pauses']
    else: # if it is not a pause
        n = 0
        a = self.result['nb_of_tot_char']
        if a > 1:
            n = self.result['nb_of_pauses'] +1
        self.result['nb_of_p_bursts'] = n
    return self.result['nb_of_p_bursts']

def number_r_bursts(self):
    """
    formula: If the last event is revision then number of r-bursts == number of revisions.
    else number of r-bursts == number of revisions + 1
    :return: the number of r-bursts(int)
    """
    r = 0
    self.result['nb_of_r_bursts'] = 0
    dt1 = self.contents[(self.contents['event_type_t'] == 'End edit')]
    v = dt1['revisions'].values[-1]
    if v == True: # if the last event is a revision
        self.result['nb_of_r_bursts'] = self.result['nb_of_revisions']
     # the number of r-bursts equals the number of revisions
    else:
        a = self.result['nb_of_del_char']
        if a == 0: # if there are no revisions
            r = 1 #there will be one revision burst
        else:
            r = a + 1 # otherwise it is the number of revisions + 1
        self.result['nb_of_r_bursts'] = r
    return self.result['nb_of_r_bursts']

def number_of_words(self):
    """
    formula: sum (single words)
    :return: the number of words in the message(int)
    """
    self.result['nb_of_words'] = 0
    if len(self.contents[(self.contents['event_type_t'] == 'Text change')]) > 1:
     # we cannot take the previous line of the last line if there is no previous line.
    # Otherwise there is an IndexOutOfBounds exception.
        l = self.contents[(self.contents['event_type_t'] == 'Text change')].iloc[-2]
        tokenizer = RegexpTokenizer(r'\w+') # we use the tokenizer from NLTK library.
        text = str(l['msg_text'])
        tokens = tokenizer.tokenize(text)
        if len(tokens) != 0:
            self.result['nb_of_words'] = len(tokens)
    return self.result['nb_of_words']

"""
OUTCOME VARIABLES

We compute outcome variables (16 fluency measures that we have chosen)
"""
```

```
"""
speed fluency
"""


def tot_char_user_mess_dur(self):
    """
    formula: total characters/ user message duration
    :return: the number of characters per user message duration(float)
    """
    self.result['tot_char_user_mess_duration'] = 0
    umd = self.result['us_message_duration'].total_seconds()
    if umd != 0:
        acumd = self.result['nb_of_tot_char']/umd
        self.result['tot_char_user_mess_duration'] = round(acumd,3)
    return self.result['tot_char_user_mess_duration']


def tot_char_user_turn_dur(self):
    """
    formula: total characters/ user turn duration
    :return: the total number of characters per user turn duration(float)
    """
    self.result['tot_char_us_turn_duration'] = 0
    utd = self.result['us_turn_duration'].total_seconds()
    if utd != 0:
        acutd = self.result['nb_of_tot_char']/ utd
        self.result['tot_char_us_turn_duration'] = round(acutd,3)
    return self.result['tot_char_us_turn_duration']


def fin_char_us_mess_dur(self):
    """
    formula: final characters/ user message duration
    :return: the number of characters per user message duration (float)
    """
    self.result['final_char_us_mess_duration'] = 0
    umd = self.result['us_message_duration'].total_seconds()
    if umd != 0:
        fcum = self.result['nb_of_fin_char']/umd
        self.result['final_char_us_mess_duration'] = round(fcum,3)
    return self.result['final_char_us_mess_duration']

def fin_char_us_turn_dur(self):
    """
    formula: final characters/ user turn duration
    :return: the number of characters divided by user turn duration(float)
    """
    self.result['final_char_us_turn_dur'] = 0
    utd = self.result['us_turn_duration'].total_seconds()
    if utd != 0:
        fcut = self.result['nb_of_fin_char']/utd
        self.result['final_char_us_turn_dur'] = round(fcut,3)
    return self.result['final_char_us_turn_dur']


def words_mess_dur(self):
    """
    formula: number of words/ user message duration
    :return: the number of words per user message duration (float)
    """
    self.result['nb_words_us_mess_duration'] = 0
    umd = self.result['us_message_duration'].total_seconds()
    if umd != 0:
        wad = self.result['nb_of_words']/umd
        self.result['nb_words_us_mess_duration'] = round(wad,3)
    return self.result['nb_words_us_mess_duration']

def words_turn_dur(self):
    """
```

```
        formula: number of words/ user turn duration
        :return: the number of words per user turn duration (float)
        """
        self.result['nb_words_us_turn_duration'] = 0
        utd = self.result['us_turn_duration'].total_seconds()
        if utd != 0:
            wad = self.result['nb_of_words']/utd
            self.result['nb_words_us_turn_duration'] = round(wad,3)
        return self.result['nb_words_us_turn_duration']

    """
    breakdown fluency
    """

    def mean_pause_duration(self):
        """
        formula: duration of pauses/ number of pauses
        :return: the duration of pauses divided by the number of pauses(seconds)
        """
        self.result['mean_pause_duration'] = d_t.timedelta(microseconds=0)
        np = self.result['nb_of_pauses']
        if np != 0:
            pd = self.result['pauses_duration']/ np
            self.result['mean_pause_duration'] = round(pd.total_seconds(),3)
        return self.result['mean_pause_duration']

    def percentage_pause_time(self):
        """
        formula: (pauses duration / user message duration) * 100
        :return: the percentage of pause time in user message writing time(percent)
        """
        self.result['percentage_pause'] = 0
        p = self.result['pauses_duration'].total_seconds()
        umt = self.result['us_message_duration'].total_seconds()
        if p != 0:
            if umt != 0:
                pp = p/umt
                pps = pp * 100
                self.result['percentage_pause'] = int(pps)
        return self.result['percentage_pause']

    """
    repair fluency
    """

    def revision_episodes_mess_dur(self):
        """
        formula: number of revisions/ user message duration
        :return: the number of revisions per user message duration (float)
        """
        self.result['revisions_us_mess_duration'] = 0
        umd = self.result['us_message_duration'].total_seconds()
        if umd != 0:
            rat = self.result['nb_of_revisions']/umd
            self.result['revisions_us_mess_duration'] = round(rat,3)
        return self.result['revisions_us_mess_duration']

    def revision_episodes_turn_dur(self):
        """
        formula: number of revisions/ user turn duration
        :return: number of revisions per user turn duration(float)
        """
        self.result['revisions_us_turn_duration'] = 0
        utd = self.result['us_turn_duration'].total_seconds()
        if utd != 0:
            rat = self.result['nb_of_revisions']/utd
            self.result['revisions_us_turn_duration'] = round(rat,3)
        return self.result['revisions_us_turn_duration']

    def revision_episodes_per_words(self):
        """
        formula: number of revisions/ number of words
```

```
        :return: number of revisions per number of words(float)
        """
        self.result['revisions_per_words'] = 0
        w = self.result['nb_of_words']
        r = self.result['nb_of_revisions']
        if w != 0:
            repw = r/w
            self.result['revisions_per_words'] = round(repw,3)
        return self.result['revisions_per_words']


    def revision_proportion(self):
        """
        formula: final characters / total characters
        :return: number of final characters per total number of characters(float)
        """
        self.result['revision_proportion'] = 0
        nbac = self.result['nb_of_tot_char']
        if nbac != 0:
            rp = self.result['nb_of_fin_char']/ nbac
            self.result['revision_proportion'] = round(rp,3)
        return self.result['revision_proportion']

    """
    automatization
    """


    def mean_duration_p_bursts(self):
        """
        formula: duration of p-bursts/ number of p-bursts
        :return: duration of p-bursts per number of p-bursts(seconds)
        """
        self.result['mean_duration_p_bursts'] = d_t.timedelta(microseconds=0)
        nbp = self.result['nb_of_p_bursts']
        if nbp != 0:
            pbd = self.result['p_bursts']/ nbp
            self.result['mean_duration_p_bursts'] = round(pbd.total_seconds(),3)
        return self.result['mean_duration_p_bursts']

    def mean_duration_r_bursts(self):
        """
        formula: duration of r-bursts/ number of r-bursts
        :return: the duration of r-bursts divided by the number of r-bursts (seconds)
        """
        self.result['mean_duration_r_bursts'] = d_t.timedelta(microseconds=0)
        nbr = self.result['nb_of_r_bursts']
        if nbr != 0:
            rbd = self.result['r_bursts']/nbr
            self.result['mean_duration_r_bursts'] = round(rbd.total_seconds(),3)
        return self.result['mean_duration_r_bursts']


    def mean_length_p_bursts(self):
        """
        formula: total number of characters/ number of p-bursts
        :return: total number of characters per number of p-bursts(float)
        """
        self.result['mean_length_of_p_bursts'] = 0
        nbp = self.result['nb_of_p_bursts']
        if nbp != 0:
            mlpb = self.result['nb_of_tot_char']/ nbp
            self.result['mean_length_of_p_bursts'] = round(mlpb,3)
        return self.result['mean_length_of_p_bursts']

    def mean_length_r_bursts(self):
        """
        formula: total number of characters/ number of r-bursts
        :return: total number of characters per number of r-bursts(float)
        """
        self.result['mean_length_of_r_bursts'] = 0
        nbr = self.result['nb_of_r_bursts']
```

```
        if nbr != 0:
            mlrb = self.result['nb_of_tot_char']/nbr
            self.result['mean_length_of_r_bursts'] = round(mlrb,3)
        return self.result['mean_length_of_r_bursts']
```

c) Class TurnFactory

```
"""
Author: Daria Alexander

The class TurnFactory divides data into turns and computes fluency measures on the level of each
turn by creating objects of the class MessageHandling, one of the parameters of which are
contents of each turn extracted from Pandas dataframe and applying the methods of the class
MessageHandling to those objects. It stores the results of the calculations in the dictionary
result. The results are also written in a CSV file.
"""

import csv
import pandas as pd
import numpy as nm

# we import the class eventLogLoader
import eventLogLoader as logload
import messageHandling as mymessage
from datetime import datetime
from datetime import timedelta

class turnFactory():

    #constructor
    def __init__(self):
        self.data = None # Pandas dataframe
        self.messagelist =  {} # a dictionary of messages

    # initialise the methods from EventLogLoader

    def initialiseFactory(self, path_json):
        """
        :param path_json:
        initialises turn factory by creating an object of a class EventLogLoader
        """
        try:
            initializer = logload.eventLogLoader()
            initializer.load(path_json)
            initializer.extract_length_diff()
            initializer.extract_pauses()
            initializer.extract_revisions()
            self.data = initializer.get_events()
        except (FileNotFoundError, IOError):
            print("File not found")

    def getTurns(self):
        return

    def identifyTurns(self):
        """
        computes fluency measures by creating objects of the class MessageHandling
        writes unique IDs as keys in the dictionary messagelist and the objects
         of the class MessageHandling as values, computes fluency measures on the level
         of each turn and writes them in the dictionary result of the class MessageHandling,
         writes the results of the computation in a CSV file.
         """
         # We obtain series, that contain the messages unique ID's
         turnId_series = self.data['msg_id'].unique()
         #a loop over all the turn id's
               for turnId in turnId_series:
                   #we transfer the contents to the messageHandling class
                   if not nm.isnan(turnId): # if a turn id exists
                       contents = self.data.loc[self.data['msg_id'] == turnId] # we divide
                          #the data into turns
                       currentmessage = mymessage.messageHandling(contents, turnId)
```

```
            # and create objects of the class MessageHandling where the parameters
            # are contents of each turn and turn IDs
        self.messagelist[turnId] = currentmessage
            # objects of the class MessageHandling
        #become values of the dictionary messagelist.

load = logload.eventLogLoader()
b = datetime(1,1,1)
previousmessage = None
with open ('results.csv', 'w') as csv_file:
   # open a CSW file to write results there
    writer = csv.writer(csv_file, delimiter=';')
    need_write_fields = True
    for k, single_instance in self.messagelist.items():
            # we loop over all the keys in the dictionary messagelist
        single_instance.previousmessage = previousmessage
            # single_instance belongs to the messageHandling class
        previousmessage = single_instance

    """
    Basic variables

    we call the methods of the class MessageHandling and
    compute basic variables on the level of each turn.
    Those basic variables are necessary for the calculations
    """


            sub_t = single_instance.user_message_submit_time()
            first_typ = single_instance.first_typing_event()
            rec_t = single_instance.system_message_reception_time()
            us_mess_dur = single_instance.user_message_duration()
            turn_paus_dur = single_instance.user_turn_pause_duration()
            us_turn_dur = single_instance.user_turn_duration()
            pause_dur = single_instance.pauses_duration()
            p_bursts = single_instance.extract_p_bursts()
            revisions = single_instance.revisions_duration()
            r_bursts = single_instance.extract_r_bursts()
            nb_tot_char = single_instance.number_of_total_characters()
            nb_del_char = single_instance.number_of_deleted_characters()
            nb_fin_cha = single_instance.number_of_final_characters()
            nb_words = single_instance.number_of_words()
            nb_rev = single_instance.number_of_revisions()
            nb_words = single_instance.number_of_words()
            nb_pauses = single_instance.number_of_pauses()
            nb_p_bursts = single_instance.number_p_bursts()
            nb_r_bursts = single_instance.number_r_bursts()

            """
            Outcome variables

            We call the methods of the class MessageHandling
            and compute the fluency measures on the level
            of each turn
            """

            tot_char_us_mess_dur = single_instance.tot_char_user_mess_dur()
            tot_char_us_turn_dur = single_instance.tot_char_user_turn_dur()
            fin_char_mess_dur = single_instance.fin_char_us_mess_dur()
            fin_char_turn_dur = single_instance.fin_char_us_turn_dur()
            words_mess_dur = single_instance.words_mess_dur()
            words_turn_dur = single_instance.words_turn_dur()
            mean_p_duration = single_instance.mean_pause_duration()
            p_percent = single_instance.percentage_pause_time()
            rev_ep_mess_dur = single_instance.revision_episodes_mess_dur()
            rev_ep_turn_dur = single_instance.revision_episodes_turn_dur()
            rev_ep_w = single_instance.revision_episodes_per_words()
            rev_prop = single_instance.revision_proportion()
            mean_p_bursts = single_instance.mean_duration_p_bursts()
            mean_r_bursts = single_instance.mean_duration_r_bursts()
            mean_l_p_butsts = single_instance.mean_length_p_bursts()
```

```
                        mean_l_r_bursts = single_instance.mean_length_r_bursts()

                        """
                        printing the test messages
                        """

                        if single_instance.messagenumber == 12269:
                                print('message', single_instance.messagenumber,\
                                'result', single_instance.result)
                        if single_instance.messagenumber == 41108:
                                print('message', single_instance.messagenumber,\
                                'result', single_instance.result)
                        if single_instance.messagenumber == 42398:
                                print('message', single_instance.messagenumber,\
                                'result', single_instance.result)

                        """
                        writing results into a file

                        """

                        writer = csv.writer(csv_file, delimiter=';')
                        writer.writerow(['event_id'] + (list(single_instance.result.keys())))
                        list_val = [single_instance.messagenumber] # the list of keys
                        for key, value in single_instance.result.items():
                                if type(value)==type(pd._libs.tslibs.timestamps.Timestamp(0)):
                                        list_val.append(value)
                                elif type(value) == \
                                type(pd._libs.tslibs.timestamps.Timedelta(0)):
                                # transform timedelta into seconds
                                        list_val.append(value.value/10**9) # nano seconds
                                else:
                                        list_val.append(value)
                                # if not timedelta, append the value the way it is
                        writer.writerow(list_val)


if __name__ == '__main__' :
    matches = turnFactory()
    matches.initialiseFactory('logs_events.json') #initialize the factory using the CSV file
    matches.getTurns()
    matches.identifyTurns()
```