```
#!/usr/bin/python
# -*- coding: UTF-8 -*-
#
#The MIT License
#
#Copyright (c) 2011
#
#Permission is hereby granted, free of charge, to any person obtaining a copy
#of this software and associated documentation files (the "Software"), to deal
#in the Software without restriction, including without limitation the rights
#to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
#copies of the Software, and to permit persons to whom the Software is
#furnished to do so, subject to the following conditions:
#
#The above copyright notice and this permission notice shall be included in
#all copies or substantial portions of the Software.
#
#THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
#IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
#FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
#AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
#LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
#OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN
#THE SOFTWARE.
#
#Authors:
#   Vili Auvinen (vili.k.auvinen@jyu.fi)
#   Olli Kauppinen (olli.kauppinen@jyu.fi)
#   Juho Tammela (juho.i.tammela@jyu.fi)

'''
The module provides the methods for inspecting docx files.

@author: Vili Auvinen, Juho Tammela
'''

from common_methods import *
from math import fabs
from conversions import convertTwipToCm, convertTwipToPt

def _getStyleElementById(styleId, styleXml):
    '''Gets a style element by the style id from the styles.xml.

    The style Id links a paragraph using a style in document.xml to the right style in styles.xml.
    Style id can be in different languages depending on what language the Word was that wrote the document.

    @note: XML example:

    <w:p>
    <w:pPr>
    <w:pStyle w:val="Otsikko1"/> (This is the style id.)
    </w:pPr>
    <r>
    ...
    </r>
    </w:p>


    @param styleId: The style id.
    @param styleXml: styles.xml as a DOM tree.

    @return: The style element with a given style id or None if no matching style element was found.
    '''

    #styleElements = _getElementsWithinElement(styleXml, 'w:style')
    styleElements = styleXml.getElementsByTagName('w:style')
    for element in styleElements:
        if (element.getAttribute('w:styleId') == styleId):
            return element

    #errors.append(styleId + ' -style id is not found.')
    return None

def _getStyleElementByName(styleName, styleXml):
    '''Gets a style element by a style name from styles.xml.

    Style name is found in the styles.xml.
    A style has always the same name regardless of the language of the Word that wrote the document.

    @note: XML example:

    <w:style w:type="paragraph" w:styleId="Otsikko1"> (This is the style id.)
    <w:name w:val="heading 1"/> (Here is the style name.)
    ...
    </w:style>
```

```
    @param styleName: The style name.
    @param styleXml: styles.xml as a DOM tree.

    @return: The style-element with a given style name, or None if no matching style element was found.
    '''
    #nameElements = _getElementsWithinElement(styleXml, 'w:name') # w:name -element is style-element's child
    nameElements = styleXml.getElementsByTagName('w:name')
    for element in nameElements:
        if (element.getAttribute('w:val') == styleName):
            return element.parentNode
    return None

def _getBasedOnStyleId (styleName, styleXml):
    '''Get the based-on style style id for a given style from styles.xml.

    @param styleName: The style name of the style that's based-on style id is wanted.
    @param styleXml: styles.xml as a DOM tree.

    @return: The id of the based on style for a given style name, or None if there was no based on style.'''
    # FIXME: returns actually the styleId of the basedOn-style NOT the styleName.
    if _getStyleElementByName(styleName, styleXml) is not None:
        try:
            return _getStyleElementByName(styleName, styleXml).getElementsByTagName('w:basedOn')[0].getAttribute('w:val')
        except:
            return None
    return None

def _getStyleName(styleElement):
    '''Get a style name of the style element.

    @param styleElement: The style element whose style name is wanted.

    @return: The style name of a given style element.'''
    # TODO: try except
    return styleElement.getElementsByTagName('w:name')[0].getAttribute('w:val')

def _getStyleNameById(styleId, styleXml):
    '''Get the name of a style with a given style id.

    @param styleId: The style id to be looked for.
    @param styleXml: styles.xml as a DOM tree.

    @return: The style name of the style with the correct style id, or None if it wasn't found.
    '''
    styleElement = _getStyleElementById(styleId, styleXml)
    if styleElement is not None:
        return _getStyleName(styleElement)
    return None

def _getStyleIdByName(styleName, styleXml):
    '''Get the id of a style with a given style name from styles.xml.

    @param styleName: The style name to be looked for.
    @param styleXml: styles.xml as a DOM tree.

    @return: The style id of the style with the correct style name, or None if it wasn't found.
    '''
    #styleElements = _getElementsWithinElement('w:name', styleXml)
    styleElements = styleXml.getElementsByTagName('w:name')
    for styleElement in styleElements:
        if (styleElement.getAttribute('w:val').lower() == styleName.lower()):
            styleId = styleElement.parentNode.getAttribute('w:styleId')
            return styleId
    return None

def _getParagraphStyleId(p):
    '''Gets the style id of a paragraph element.

    @param p: The paragraph element.

    @return: The style id if it was found, otherwise returns None.
    '''
    try:
        styleId = p.getElementsByTagName('w:pStyle')[0].getAttribute('w:val')
    except:
        return None
    return styleId


def _getThemeFont(themeXml, styleDefinitions, themeFont):
    '''Gets a themefont from theme1.xml.

    @note: XML example:

    <a:fontScheme name="Office">
```

```
    <a:majorFont>
    <a:latin typeface="Cambria"/>
    <a:ea typeface=""/>
    <a:cs typeface=""/>
    </a:majorFont>
    <a:minorFont>
    <a:latin typeface="Calibri"/>
    <a:ea typeface=""/>
    <a:cs typeface=""/>
    </a:minorFont>
    </a:fontScheme>

    @param themeXml: theme1.xml as DOM tree.
    @param styleDefinitions: The style definitions dict.
    @see: _getCompleteStyleDefinitions.
    @param themeFont: The theme font. Should be either 'majorFont' or 'minorFont'.

    @return: The style definitions with or without changes.

    '''
    if themeFont == "" or themeFont is None:
        return styleDefinitions

    fontElement = None

    if themeFont.startswith('major'):
        fontElement = themeXml.getElementsByTagName('a:majorFont')[0]
    elif themeFont.startswith('minor'):
        fontElement = themeXml.getElementsByTagName('a:minorFont')[0]

    if fontElement is not None:
        themeFont = fontElement.getElementsByTagName('a:latin')[0].getAttribute('typeface')
        if themeFont.strip() != "":
            styleDefinitions['w:ascii'] = themeFont
            styleDefinitions['w:asciiTheme'] = None

    return styleDefinitions

def _getStyleDefinitions (element, styleDefinitions):
    ''' Return style definitions of a given element.
    First checks if the element has any children and uses recursion if some are found.
    Next checks if the element has attributes.
      - If the attribute name is a key in the dict, stores the value of the attribute.
      - If the attribute name is 'w:val' and the element tag name is a key in the dict, stores the value of the attribute.
    If the element tag name is a key in the dict and the element doesn't have any attributes or children, stores value '1' i

    @param element: Style definitions are searched inside this element
    @param styleDefinitions: A dict where the style definitions are stored.
                             May contain tag names or attribute names and some default values.

    @return: The style definitions in a dict.
    '''

#    <w:pPr>
#        <w:pBdr>
#            <w:top w:val="single" w:sz="24" w:space="0" w:color="4F81BD"/>
#            <w:left w:val="single" w:sz="24" w:space="0" w:color="4F81BD"/>
#            <w:bottom w:val="single" w:sz="24" w:space="0" w:color="4F81BD"/>
#            <w:right w:val="single" w:sz="24" w:space="0" w:color="4F81BD"/>
#        </w:pBdr>
#        <w:shd w:val="clear" w:color="auto" w:fill="4F81BD"/>
#        <w:spacing w:before="360" w:after="0"/>
#        <w:outlineLvl w:val="0"/>
#    </w:pPr>
# We are not interested in border style definitions as above at the moment. Implement here if needed later.
    if element.tagName == "w:pBdr":
        return styleDefinitions

    for child in element.childNodes:
        if child.nodeType != child.TEXT_NODE:
            styleDefinitions = _getStyleDefinitions(child, styleDefinitions)

    if element.hasAttributes():
        for i in range (0, element.attributes.length):
            attributeName = element.attributes.item(i).name
            if styleDefinitions.has_key(attributeName):
                styleDefinitions[attributeName] = element.attributes.item(i).value
#            if attributeName != "w:val":
#                styleDefinitions[element.attributes.item(i).name] = element.attributes.item(i).value
            elif styleDefinitions.has_key(element.tagName) and attributeName == "w:val":
                styleDefinitions[element.tagName] = element.attributes.item(i).value

    elif styleDefinitions.has_key(element.tagName) and element.hasChildNodes() is False:
        styleDefinitions[element.tagName] = True

    return styleDefinitions
```

```
def getStyle(document, requirementStyleName):
    '''Gets all definitions of a style from document dictionary.

    Converts twips to centimeters.

    @return: A dict with all the style definitions of the one style with the
             translated keys to match return value odt_inspector's getStyle(). False, if the style was not found.
    '''
    styleXml = document['word/styles.xml']
    themeXml = document['word/theme/theme1.xml']


    if _isStyleUsed(document, requirementStyleName) is False:
        return False

    #styleName is a capitalized string or CamelCase string (for example: Normal, Body Text)
    #However, heading stylenames are in the xml in lower case (heading 1)
    #Also stylenames such as toc 1, index 1, footer, header, caption are in lower case.

    #The following fixes this broblem, as it gets the styleId by comparing lowercase stylename strings,
    #and then gets the correct styleName by the styleId.
    styleId = _getStyleIdByName(requirementStyleName, styleXml)
    styleName = _getStyleNameById(styleId, styleXml)

    styleDefinitions = _getCompleteStyleDefinitions(styleXml, styleName, themeXml)

    if styleDefinitions is None:
        return False

    translateDict = {'w:name':'styleName',
        'w:ascii':'fontName',
        'w:sz':'fontSize',
        'w:caps':'transform',
        'w:left':'indentLeft',
        'w:right':'indentRight',
        'w:firstLine':'indentFirstLine',
        'w:line':'linespacing',
        'w:before':'spacingBefore',
        'w:after':'spacingAfter',
        'w:keepNext':'keepWithNext',
        'w:jc':'alignment',
        'w:widowControl':'widowControl',
        #'w:widowControl':'widows', #TODO: widows kovakoodattuna
        'w:b':'bold',
        'w:i':'italic'}
    styleDict = {}

    # In case styleName is changed, let's return the same styleName as was given in the parameter.
    styleDefinitions['w:name'] = requirementStyleName
    # Line spacing single (1) = 12 points = 240 twips.
    # <w:spacing w:before="300" w:after="340" w:line="240" w:lineRule="auto"/>
    # For example: w:line="360" -> line spacing is 1.5
    styleDefinitions['w:line'] = float(styleDefinitions['w:line']) / float(240.0)

    styleDefinitions['w:sz'] = round(float(styleDefinitions['w:sz']) / 2, 1) # nyt voi olla 13.5
    styleDefinitions['w:before'] = convertTwipToPt(float(styleDefinitions['w:before']))
    styleDefinitions['w:after'] = convertTwipToPt(float(styleDefinitions['w:after']))
#FIXME:
#    File "/var/www/virtual.hosts/sovellusprojektit.it.jyu.fi/parsi/sovellus/docx_inspector.py", line 244, in getStyle
#    styleDefinitions['w:left'] = round( convertTwipToCm( float(styleDefinitions['w:left']) ), 1)
#    ValueError: invalid literal for float(): single
    styleDefinitions['w:left'] = round(convertTwipToCm(float(styleDefinitions['w:left'])), 1)
    # Rounded with a precision of one decimal. If this is used more often, update to conversions.py.

    for key in translateDict.keys():
        styleDict[translateDict[key]] = styleDefinitions[key]

    return styleDict

def _getCompleteStyleDefinitions(styleXml, styleName, themeXml):
    ''' Returns the style definition of the given style from style.xml and theme1.xml.
    Recursion used because the style can be based on some other style.
    In addition, the base style gets style definitions from the document defaults.
    Finally, some style definitions are not found in the XML file at all. These definitions use some default value which mus

    @note: XML example:

    <w:style w:type="paragraph" w:default="1" w:styleId="Normaali">
    <w:name w:val="Normal"/>
    <w:qFormat/>
    <w:rsid w:val="006B493C"/>
    <w:pPr>
    <w:spacing w:before="140" w:after="220" w:line="360" w:lineRule="auto"/>
    <w:ind w:left="567"/>
```

```
    <w:jc w:val="both"/>
    </w:pPr>
    <w:rPr>
    <w:rFonts w:ascii="Georgia" w:hAnsi="Georgia"/>
    <w:lang w:val="fi-FI"/>
    </w:rPr>
    </w:style>

    @param styleXml: styles.xml-file as a DOM tree.
    @param styleName: The name of the style (NOT the id)
    @see: _getStyleElementById and _getStyleElementByName for difference.
    @param themeXml: theme1.xml as DOM tree.

    @return: Style definitions in a dict.
    '''

#     If fontSize is not found mentioned in xml, the user has used the default size which is 12 (*2 = 24) for text body
#     TODO: initialize dict with the default style definitions.
#     TODO: some of the values are in twips, convert to cm.

#      Complete style-element specification can be found at: http://www.schemacentral.com/sc/ooxml/e-w_style-1.html
#      Style-element content for the most part:
#      w:name [0..1]    Primary Style Name
#      w:aliases [0..1]    Alternate Style Names
#      w:basedOn [0..1]    Parent Style ID
#      w:next [0..1]    Style For Next Paragraph
#      w:link [0..1]    Linked Style Reference
#      w:autoRedefine [0..1]    Automatically Merge User Formatting Into Style Definition
#      w:hidden [0..1]    Hide Style From User Interface
#      w:semiHidden [0..1]    Hide Style From Main User Interface
#      w:unhideWhenUsed [0..1]    Remove Semi-Hidden Property When Style Is Used
#      w:qFormat [0..1]    Primary Style
#      w:locked [0..1]    Style Cannot Be Applied
#     w:pPr: - paragraph properties
#         w:pStyle [0..1]    Referenced Paragraph Style
#          w:keepNext [0..1]    Keep Paragraph With Next Paragraph
#          w:keepLines [0..1]    Keep All Lines On One Page
#          w:pageBreakBefore [0..1]    Start Paragraph on Next Page
#          w:widowControl [0..1]    Allow First/Last Line to Display on a Separate Page
#          w:numPr [0..1]    Numbering Definition Instance Reference
#              w:numPr/w:numId [0..1]    Numbering Definition Instance Reference
#          w:spacing [0..1]    Spacing Between Lines and Above/Below Paragraph
#             w:before    [0..1]    Spacing Above Paragraph
#             w:beforeLines    [0..1]    Spacing Above Paragraph IN Line Units
#             w:beforeAutospacing    [0..1]    Automatically Determine Spacing Above Paragraph
#             w:after    [0..1]    Spacing Below Paragraph
#             w:afterLines    [0..1]    Spacing Below Paragraph in Line Units
#             w:afterAutospacing    [0..1]    Automatically Determine Spacing Below Paragraph
#             w:line    [0..1]    Spacing Between Lines in Paragraph
#             w:lineRule    [0..1]    Type of Spacing Between Lines
#          w:ind [0..1]    Paragraph Indentation
#             w:left    [0..1]    Left Indentation
#             w:leftChars    [0..1]    Left Indentation in Character Units
#             w:right    [0..1]   Right Indentation
#             w:rightChars    [0..1]    Right Indentation in Character Units
#             w:hanging    [0..1]    Indentation Removed from First Line
#             w:hangingChars    [0..1] Indentation Removed From First Line in Character Units
#             w:firstLine    [0..1] Additional First Line Indentation
#             w:firstLineChars    [0..1] Additional First Line Indentation in Character Units
#          w:jc [0..1]    Paragraph Alignment
#          w:outlineLvl [0..1]    Associated Outline Level
#     w:rPr - run properties
#         2. w:rFonts [0..1]    Run Fonts
#                 w:ascii    [0..1]    w:ST_String    ASCII Font
#                 w:hAnsi    [0..1]    w:ST_String    High ANSI Font
#                 w:cs    [0..1]    w:ST_String    Complex Script Font
#                 w:asciiTheme    [0..1]    w:ST_Theme    ASCII Theme Font
#                 w:hAnsiTheme    [0..1]    w:ST_Theme    High ANSI Theme Font
#                 w:cstheme    [0..1]    w:ST_Theme    Complex Script Theme Font
#          3. w:b [0..1]    Bold
#          4. w:bCs [0..1]    Complex Script Bold
#          5. w:i [0..1]    Italics
#          6. w:iCs [0..1]    Complex Script Italics
#          7. w:caps [0..1]    Display All Characters As Capital Letters
#          8. w:smallCaps [0..1]    Small Caps
#          9. w:strike [0..1]    Single Strikethrough
#         10. w:dstrike [0..1]    Double Strikethrough
#         11. w:outline [0..1]    Display Character Outline
#         15. w:noProof [0..1]    Do Not Check Spelling or Grammar
#         24. w:sz [0..1]    Font Size
#         25. w:szCs [0..1]    Complex Script Font Size
#         27. w:u [0..1]    Underline
#         36. w:lang [0..1]    Languages for Run Content

#      Initialize the styleDefinitions dict with the style definitions you wish to search and their default values.
#      Rules of initialization:
```

```
#     1. Use attribute name as a key.
#     2. If attribute name is 'w:val', use element tag name as a key.
#     3. If the element can be empty and attributes are optional, use element tag name as a key.
    styleDefinitions = {'w:name': None,
                        'w:basedOn': None,
                        'w:next': None,
                        'w:link': None,
                        'w:autoRedefine': None,
                        'w:left': '0',
                        'w:right': '0',
                        'w:firstLine': None,
                        'w:line': '240.0',
                        'w:before': '0',
                        'w:after': '0',
                        'w:widowControl': True, # on as default
                        'w:jc': 'left', #left as default
                        'w:sz': '24',
                        'w:ascii': None,
                        'w:asciiTheme': None,
                        'w:b': False, # default
                        'w:i': False, # default
                        'w:u': False, # default
                        #'w:outline': None,
                        #'w:numId': None,
                        #'w:ilvl': None,
                        'w:lang': None,
                        'w:keepNext': None,
                        'w:keepLines': None,
                        'w:pageBreakBefore': None,
                        'w:caps': None}

    styleElement = _getStyleElementByName(styleName, styleXml)

    if styleElement is None:
        return

    basedOnStyleId = _getBasedOnStyleId(styleName, styleXml)

#     Recursion: if this style has a basedOn-style, method calls itself with the basedOnStyleName.
#     When style has no basedOn-style, get the definitions in w:docDefaults-element first.
    if basedOnStyleId is not None:
        styleDefinitions = _getCompleteStyleDefinitions(styleXml, _getStyleNameById(basedOnStyleId, styleXml), themeXml)
    elif basedOnStyleId is None:
        styleDefinitions = _getStyleDefinitions(styleXml.getElementsByTagName('w:docDefaults')[0], styleDefinitions)

    styleDefinitions = _getStyleDefinitions(styleElement, styleDefinitions)

    if(styleDefinitions.has_key('w:asciiTheme')):
        styleDefinitions = _getThemeFont(themeXml, styleDefinitions, styleDefinitions['w:asciiTheme'])

    return styleDefinitions

def _getElementValueWithinElement(elementTagName, element):
    '''Gets the text content of the first element with a certain tag in the given DOM tree.

    @return: The text value of the element, or None if something went wrong.
    '''
    try:
        value = element.getElementsByTagName(elementTagName)[0].firstChild.nodeValue
        return value
    except:
        return None

def _getElementWithinElement(element, elementTagName):
    '''Gets the first child of an element with the given tag name.

    Returns the element of a the given parent element with the given elementTagName.

    @param element: The element whose children are searched.
    @param elementTagName: The tag name of the wanted element.

    @return: An element with the right tag name, or None if it wasn't found.
    '''
    try:
        value = element.getElementsByTagName(elementTagName)[0]
    except IndexError:
        return None
    return value

def _getElementsWithinElement(element, elementTagName):
    '''Gets the children of an element with the given tag name.

    Returns the element of the given parent element by the given elementTagName.

    @param element: The element whose children are searched.
    @param elementTagName: Tag name of the wanted elements.
```

```
    @return: The list of elements with the right tag name, or None if none was found.
    '''
    elements = element.getElementsByTagName(elementTagName)
    if len(elements) == 0:
        return None
    return elements

#def _getElementsWithinElement(elementTagName, xmlData):
#    ''' Returns the elements of the given xml file if xmlData and the given elementTagName
#        exist in the xmlData.
#
#       xmlData - the given xml file'''
#    elements = xmlData.getElementsByTagName(elementTagName)
#    if len(elements) == 0:
#        return None
#    return elements


def _getTargetXmlFileByHeader(header, document):
    '''Gets a header reference target xml-file as a DOM tree.'''
    #TODO: try except
    targetFile = getRelsTargetByRId(header.getAttribute('r:id'), document['word/_rels/document.xml.rels'])
    targetFileXml = document['word/' + targetFile] # omaan metodiin
    #targetFileXml = minidom.parseString(targetFile)
    return targetFileXml

def _checkFrontPageHeadersAndFooters(references, document):
    '''Goes through header or footer references and checks if there is any content in them.

    Checks if there are headers or footers in the front page by looking for <w:t> tags.
    Even if there are references to headers or footers, they might be empty.

    @param references: Header or footer references.

    @return: The header or footer target XML file as a DOM tree, or None if no headers or footers were found.
    '''

    if references is not None:
        for header in references:
            targetFileXml = _getTargetXmlFileByHeader(header, document)
            #TODO: eksaktimmin, _getElementValueWithinElement?
            if _getElementsWithinElement(targetFileXml, 'w:t') is not None:
                return targetFileXml


def _checkAutomaticPageNumbering(section, headerReference, footerReference, document, errorIds, numStartKey):
    '''Checks if a section has an automatic page numbering and gets the numbering format.

    First goes through the section element and checks that the numbering starts at 1.
    Gets the section numbering of format definition.
    If it is defined, returns it.
    If a numbering format is not found in the section properties, it defaults to 'Standard'.
    If the numbering format is standard, checks the header and footer references for other numbering format definitions.
    The numbering format in the header or the footer reference is sometimes in <w:instrText> element inside the content of P

    @param section: The section element to be searched for.
    @param headerReference: The current header of the section element as a DOM tree.
    @param footerReference: The current footer of the section element as a DOM tree.
    @param document: The document as a dict of DOM tree pairs.
    @param errorIds: The dict for appending errors True/False.
    @param numStartKey: The key for errorIds to append numbering start error.

    @return: The page numbering as a string format, or False if there was no page numbering or the numbering was both in hea
    '''
    pgNumTypeElement = _getElementWithinElement(section, 'w:pgNumType')
    if pgNumTypeElement is None:
        return False

    startNum = pgNumTypeElement.getAttribute('w:start')
    if str(startNum) != '1':
        errorIds[numStartKey] = False
    else:
        errorIds[numStartKey] = True

    numFormat = _getPgNumFormat(pgNumTypeElement)

    #1. check if instrtext in referenceXml.
    #2. check if 'page' and 'mergeformat' texts are found -> page numbering field is found
    #3. check if there is a pagenumbering format definition in instrtext and return it
    #4. else return numFormat = "Standard"
    if numFormat == "Standard":
        headerFormat = None
        if headerReference is not None:

            elementValue = _getElementValueWithinElement('w:instrText', headerReference)
```

```
            if elementValue is None:
                headerFormat = None
            elif elementValue.find('PAGE') and elementValue.find('MERGEFORMAT'):
                splitted = elementValue.split('\*')
                if len(splitted) > 2:
                    headerFormat = splitted[1].lower().strip()

        footerFormat = None
        if footerReference is not None:

            elementValue = _getElementValueWithinElement('w:instrText', headerReference)
            if elementValue is None:
                footerFormat = None
            elif elementValue.find('PAGE') and elementValue.find('MERGEFORMAT'):
                splitted = elementValue.split('\*')
                if len(splitted) > 2:
                    footerFormat = splitted[1].lower().strip()

        if headerFormat is not None and footerFormat is not None:
            #TODO: numbering in both header and footer.
            return False
        elif headerFormat is not None and headerFormat != "Standard":
            return headerFormat
        elif footerFormat is not None and footerFormat != "Standard":
            return footerFormat

    return numFormat

def _checkNameInHeaderOrFooter(reference, document):
    '''Looks for text inside a header or footer and sees if the last modifier's name is in there.

    Problem: sometimes we want to check that there is no name in the header or the footer.
    If a name is found but it's different from the last modifier's name, result is False, even though a name is in a header/
    For now just tries to check that either the name of the last modifier or just some name was found.

    @param reference: The header or footer XML file as a DOM tree.

    @return: True if a name is found in the text, False otherwise.
    '''
    if reference is not None:
        pElements = _getElementsWithinElement(reference, 'w:p')
        for pElement in pElements:
            textContent = getTextContent(pElement)


            #FIXME: if the name is different in the text and in the settings, this could give false negatives:
            #For example, if we don't want that toc section has a name in footer or header and the name is different in the
            #and in the settings, this function returns False even though there is a name in the header or footer.
            if textContent.find(_getLastModifier(document["docProps/core.xml"])) != -1:
                return True

            #FIXME: fix the problem above, then see if this is necessary:
            #When the following if is done, the function can return True even though the name is different in the header or
            #and document settings.

            #check if the content is a digit -> page number
            #split at whitespace -> len > 1 -> probably a name!
            #check if the content is longer than 3 characters -> probably a name.
            if textContent.find('PAGE') == -1:
                strippedContent = textContent.strip()
                if strippedContent.isdigit():
                    continue
                splittedContent = strippedContent.split()
                if len(splittedContent) > 1:
                    return True
                if len(strippedContent) > 3:
                    return True

    return False


def _getPgNumFormat(sectionPgNumType):
    ''' Gets the number format of the given section page number type.

    @param sectionPgNumType: The given page number type element of the section

    @return: The numbering format, defaults to 'Standard' if nothing else is defined.
    '''
    numFormat = 'Standard'
    if sectionPgNumType:
        if sectionPgNumType.hasAttribute('w:fmt'):
            numFormat = sectionPgNumType.getAttribute('w:fmt')

    return numFormat

def checkHeadersAndFooters(document):
```

```
''' Checks that the headers and footers of a document are made correctly.

Assumes that the document has three sections:
    1. cover section
    2. table of contents section or toc section
    3. actual content section or text section

@see: checkSections method must pass in order to run this method

@note:
Places findings in the errorIds-dict as key-boolean pairs:

'frontPage': was there headers or footers in the cover section.

'tocPageNumbering': is there a page numbering in the toc section.

'differentPageNumbering': is the page numbering different in the cover and text sections.

'nameInToc': is the last modifiers name in toc section header or footer.

'nameInText': is the last modifiers name in text section header or footer.

'pageNumbering': is there a page numbering in the text section.

'tocNumStart': does the toc section page numbering start at 1.

'textNumStart': does the text section page numbering start at 1.

'titlePg': is the Microsoft Office setting "Different first page" on.

@note: XML example:

<w:pgNumType w:fmt="lowerRoman" w:start="1"/>)

<w:pgNumType w:start="1"/>

@return: Findings in the errorIds-dict as key-boolean pairs as described above.
'''

docXml = document['word/document.xml']

allSectionProperties = getSectionElementsBySections(docXml)

errorIds = {'frontPage': None, 'tocPageNumbering': None, 'differentPageNumbering': None,
        'nameInToc': None, 'nameInText': None, 'pageNumbering': None, 'tocNumStart': None,
        'textNumStart': None, 'titlePg': None} # @see XML requirement file

currentHdrRef = None
currentFtrRef = None

#allSectionProperties[0] = cover page
#allSectionProperties[1] = table of contents page
#allSectionProperties[2] = actual document section

coverSection = allSectionProperties[0]
tocSection = allSectionProperties[1]
commonSection = allSectionProperties[2]

for coverSectPr in coverSection:

    frontPageHeaderReferences = _getElementsWithinElement(coverSectPr, 'w:headerReference')
    frontPageFooterReferences = _getElementsWithinElement(coverSectPr, 'w:footerReference')

    frontPageHdrRef = _checkFrontPageHeadersAndFooters(frontPageHeaderReferences, document)
    if frontPageHdrRef is not None:
        currentHdrRef = frontPageHdrRef

    frontPageFtrRef = _checkFrontPageHeadersAndFooters(frontPageFooterReferences, document)
    if frontPageFtrRef is not None:
        currentFtrRef = frontPageFtrRef

if currentHdrRef is not None or currentFtrRef is not None:
    errorIds["frontPage"] = True
else:
    errorIds["frontPage"] = False


tocSectionNumberingFormat = None
for tocSectPr in tocSection:

    tocHdrRefs = _getElementsWithinElement(tocSectPr, 'w:headerReference')
    tocFtrRefs = _getElementsWithinElement(tocSectPr, 'w:footerReference')

    tocHdrRef = _checkFrontPageHeadersAndFooters(tocHdrRefs, document)
    if tocHdrRef is not None:
        currentHdrRef = tocHdrRef
```

```
            tocFtrRef = _checkFrontPageHeadersAndFooters(tocFtrRefs, document)
            if tocFtrRef is not None:
                currentFtrRef = tocFtrRef

            tocSectionNumberingFormat = _checkAutomaticPageNumbering(tocSectPr, currentHdrRef, currentFtrRef, document, errorIds

            if tocSectionNumberingFormat is not False:
                #PAGE NUMBERING IN HEADER AND FOOTER:
                errorIds['tocPageNumbering'] = True
                break
            else:
                errorIds['tocPageNumbering'] = False

        # is document's writer's name in tocSection header or footer?
        if _checkNameInHeaderOrFooter(currentHdrRef, document) is False and \
        _checkNameInHeaderOrFooter(currentFtrRef, document) is False:
            errorIds['nameInToc'] = False
        else: errorIds['nameInToc'] = True


        commonSectionNumberingFormat = None
        for commonSectPr in commonSection:
            commonHdrRefs = _getElementsWithinElement(commonSectPr, 'w:headerReference')
            commonFtrRefs = _getElementsWithinElement(commonSectPr, 'w:footerReference')

            commonHdrRef = _checkFrontPageHeadersAndFooters(commonHdrRefs, document)
            if commonHdrRef is not None:
                currentHdrRef = commonHdrRef

            commonFtrRef = _checkFrontPageHeadersAndFooters(commonFtrRefs, document)
            if commonFtrRef is not None:
                currentFtrRef = commonFtrRef

            commonSectionNumberingFormat = _checkAutomaticPageNumbering(commonSectPr, currentHdrRef, currentFtrRef, document, er

            if commonSectionNumberingFormat is not False:
                #PAGE NUMBERING IN HEADER AND FOOTER:
                errorIds['pageNumbering'] = True
                break
            else:
                errorIds['pageNumbering'] = False

        #  is the document's maker's name in the body part of the document
        if _checkNameInHeaderOrFooter(currentHdrRef, document) is True or \
        _checkNameInHeaderOrFooter(currentFtrRef, document) is True:
            errorIds['nameInText'] = True
        else: errorIds['nameInText'] = False

        if tocSectionNumberingFormat is not False and commonSectionNumberingFormat is not False:
            if tocSectionNumberingFormat != commonSectionNumberingFormat:
                errorIds['differentPageNumbering'] = True
            else:
                errorIds['differentPageNumbering'] = False
        else:
            errorIds['differentPageNumbering'] = False

        return errorIds

    def getParagraphElementsBySections(docXml, sectionName):
        '''Get paragraph elements of the wanted section.
        The page breaking section break elements changes section, continuous section brake elements don't change section.

        The first list of the section elements is the cover section.
        The second list of the section elements is the table of contents-section.
        The third list of the section elements is the text section.
        The document has to have at least 3 sections.

        @param docXml: The document.xml file as a DOM tree.
        @param sectionName: The wanted section can be 'cover', 'toc' or 'text'.

        @return: The list of the section elements.
        '''

        sectionList = [[]]
        #sectionList = [[w:p],[w:p,w:p],[w:p]]
        bodyElement = docXml.getElementsByTagName('w:body')[0] # always exactly 1 element

        i = 0

        for textP in bodyElement.childNodes:
            sectionList[i].append(textP)
            sectPrs = textP.getElementsByTagName('w:sectPr')
            if len(sectPrs) != 0:
                typeElement = _getElementWithinElement(sectPrs[0], 'w:type')
                if typeElement is not None:
```

```
                if typeElement.getAttribute('w:val') == 'continuous':
                    continue
            else:
                i += 1
                sectionList.append([])

    sectionElements = {'cover':sectionList[0], 'toc':sectionList[1], 'text':sectionList[2]}
    if sectionElements.has_key(sectionName):
        return sectionElements[sectionName]

def getSectionElementsBySections(docXml, index = None):
    '''Gets all the w:sectPr elements of a document or optionally the w:sectPr elements of a specific section.

    w:sectPr elements are stored in a two dimensional list.
    Continuous section breaks are appended to current outer list index.
    The page breaking section raises the outer list index.

    @param index: The index of the outer pageSections list that is get. None by default.

    @return: The two dimensional list of all w:sectPr elements if index is None. Otherwise returns the list at the given ind
    '''
    sectionElements = _getElementsWithinElement(docXml, "w:sectPr")

    pageSections = [[]]

    i = 0

    for section in sectionElements:
        typeElement = _getElementWithinElement(section, 'w:type')
        if typeElement is not None:
            if typeElement.getAttribute('w:val') == 'continuous':
                pageSections[i].append(section)
        else:
            pageSections[i].append(section)
            pageSections.append([])
            i += 1

    if len(pageSections[len(pageSections) - 1]) == 0:
        pageSections.remove([])

    if index is None:
        return pageSections
    else:
        return pageSections[index]

def _areSectionsOverlapping(outerParagraphElements, innerParagraphElements, errorList, errorMsg, expectedResult):
    '''Goes through two lists of paragraph elements checking if the same paragraph is in both lists.

    @param outerParagraphElements: The outer paragraphlist to be searched for.
    @param innerParagraphElements: The inner pagraphlist to be searched for.
    @param errorList: The list for appending error messages.
    @param errorMsg: The error message to be appended.
    @param expectedResult: Boolean of the expected result.

    @return: expectedResult changed or unchanged.
    '''
    found = False
    for coverElement in outerParagraphElements:
        if found is True:
            break
        for element in innerParagraphElements:
            if coverElement.isSameNode(element): # is the table of contents in the cover section, where it shouldn't be
                expectedResult = not expectedResult
                errorList.append(errorMsg)
                found = True
                break

    return expectedResult

def checkSections(document, errorList):
    '''Goes through the section elements in the document checking that the sections are done properly.

    There must be at least three sections in the document.
    The cover page and the table of the contents cannot be in the same section.
    Also checks that the Microsoft Office Word setting "Different first page" is off.

    @return: True if everything went well, False if something went terribly wrong or
              error list if an error was found and the checking could be completed.
    '''
    docXml = document['word/document.xml']
    styleXml = document['word/styles.xml']
    cover = True
    toc = False

    allSectionProperties = getSectionElementsBySections(docXml)
```

```
    if len(allSectionProperties) < 3:
        return False

    for section in allSectionProperties:
        for sectPr in section:
            if len(sectPr.getElementsByTagName('w:titlePg')) > 0:
                errorList.append('titlePg')
                return False

    tocParagraphs = _getParagraphElementsBySequentialStyleName('toc', styleXml, docXml)
    coverSectionParagraphs = getParagraphElementsBySections(docXml, 'cover') # cover page
    tocSectionParagraphs = getParagraphElementsBySections(docXml, 'toc')

    cover = _areSectionsOverlapping(coverSectionParagraphs, tocParagraphs, errorList, "cover", cover)

    toc = _areSectionsOverlapping(tocSectionParagraphs, tocParagraphs, errorList, "toc", toc)

    # TODO: error handling, errorIdsAndPositions
    if toc is True and cover is True:
        return True
    else:
        return errorList

#def getSectionProperties(document):
#    ''' Checks all the sectPr-elements. There must be at least three section breaks  (first page, toc-page and content) in
#        The margins should be the same throughout the whole document?
#
#        @return: section properties in a dict'''
#
#    docXml = document['word/document.xml']
#
#    finalSectionProperties = dict([['w:w', None], ['w:h', None], ['w:top', None], ['w:right', None], ['w:bottom', None], \
#                                ['w:left', None], ['w:right', None], ['w:header', None], ['w:footer', None], \
#                                ['w:gutter', None], ['w:start', None], ['w:space', None], ['w:linePitch', None], ['w:titlePg
#
#    allSectionProperties = _getElements('w:sectPr', docXml) # returns a nodeList
#
#    for element in allSectionProperties:
#        finalSectionProperties = _getStyleDefinitions(element, finalSectionProperties)
#        #TODO: conversions from twips to cm
#
#
#    return finalSectionProperties


def _checkPageProperties(allSectionProperties, pageProperties, tagName):
    ''' Goes through all section properties to see that they have coherent property values.

    If the property value is the same in all section elements, the value is stored in pageProperties.
    If something is different between the sections, it's wrong and the page property is set False.
    For example, if two different section elements have different page top marginal, the property is set False.

    @param allSectionProperties: All w:sectPr elements of the document.
    @param pageProperties: the allowed page properties are {'top': None, 'right': None, 'bottom': None, 'left': None} or {'w
    @param tagName: Tag name of the element whose properties are checked.

    @return: pageProperties dict with coherent page values and incoherent values set as False.
    '''

    for element in allSectionProperties:
        for key in pageProperties.keys():
            size = _getElementWithinElement(element, tagName).getAttribute('w:' + key)
            if pageProperties[key] is None:
                pageProperties[key] = size
            elif pageProperties[key] != size:
                pageProperties[key] = False # False means that the information has bee changed.

    return pageProperties

def _convertSizes(sizesDict):
    for key in sizesDict.keys():
        value = sizesDict[key]
        value = convertTwipToCm(int(value))
        roundedValue = round(value, 1) # pyöristetään yhden desimaalin tarkkuudella
        sizesDict[key] = roundedValue

    return sizesDict

def getPageMarginals(document):
    '''Gets the document page marginals sizes.

    @return: False if the marginals are not coherent, otherwise a dictionary containing the marginal sizes.
    '''
    allSectionProperties = _getElementsWithinElement(document['word/document.xml'], 'w:sectPr')

    pageMarginals = {'top': None, 'right': None, 'bottom': None, 'left': None}
```

```
                    # { 'header': None, 'footer': None}
                    # Header and footer sizes can be different in different sections of the document.
                    # For example if one section does not have header of footer at all and other section has them, the size

    _checkPageProperties(allSectionProperties, pageMarginals, 'w:pgMar')

    for key in pageMarginals.keys():
        if pageMarginals[key] == False:
            return False

    return _convertSizes(pageMarginals)

def getPageSize(document):
    '''Gets the document page sizes.

    @return: False if the page sizes are not coherent, otherwise a dictionary containing the page width and length.
    '''
    allSectionProperties = _getElementsWithinElement(document['word/document.xml'], 'w:sectPr')

    pageSize = {'w': None, 'h': None} # width ja heigth

    _checkPageProperties(allSectionProperties, pageSize, 'w:pgSz')

    finalPageSize = {'width': pageSize['w'], 'height':pageSize['h']}

    for key in finalPageSize.keys():
        if finalPageSize[key] == False:
            return False

    return _convertSizes(finalPageSize)


def _getTitle(coreXml):
    '''Gets the title as set in document setting, None if not found.'''
    return _getElementValueWithinElement('dc:title', coreXml)

def _getCreator(coreXml):
    '''Gets the document creator as set in document setting, None if not found.'''
#    return mso_meta_inspector._getCreator(coreXml)
    return _getElementValueWithinElement('dc:creator', coreXml)

def _getLastModifier(coreXml):
    '''Gets the document last modifier as set in document setting, None if not found.'''
    return _getElementValueWithinElement('cp:lastModifiedBy', coreXml)

def _getCreateDate(coreXml):
    '''Gets the document creatin date as found in document setting, None if not found.'''
    return _getElementValueWithinElement('dcterms:created', coreXml)

def _getLastModifiedDate(coreXml):
    '''Gets the document last modified date as found in document setting, None if not found.'''
    return _getElementValueWithinElement('dcterms:modified', coreXml)

def _getRevision(coreXml):
    '''Gets revision of the document as found in document setting, None if not found.'''
    return _getElementValueWithinElement('cp:revision', coreXml)


def _getTextFromParagraph(paragraph):
    '''Gets the text content of <w:t>-elements from the given (paragraph) element.

    @return: the text content as a string.
    '''
    eventualText = ''

    # gets all text elements from the given paragraphs
    textElements = _getElementsWithinElement(paragraph, 'w:t')
    if textElements is not None:
        for i in textElements:
            eventualText += i.firstChild.nodeValue
    return eventualText

def checkTocContent(document):
    '''Checks if all of the headings created in the document are listed in the table of contents.

    @return: True if toc matches the headings content, False otherwise.
    '''

    docXml = document['word/document.xml']
    styleXml = document['word/styles.xml']

    headingParagraphs = _getParagraphElementsBySequentialStyleName("heading", styleXml , docXml)
    #docHeadings = _getTextFromParagraph(headingParagraphs)

    docTocStyles = _getParagraphElementsBySequentialStyleName('toc', styleXml, docXml)
    #tocHeadings = _getTextFromParagraph(docTocStyles)
```

```
    docHeadings = []
    for heading in headingParagraphs:
        headingText = _getTextFromParagraph(heading).strip()
        if headingText != "":
            docHeadings.append(headingText)


    tocHeadings = []
    for tocStyle in docTocStyles:
        tocHeadings.append(_getTextFromParagraph(tocStyle).strip())

    docHeadingsLength = len(docHeadings)
    tocHeadingsLength = len(tocHeadings)
    listLength = len(docHeadings) # oletuksena docHeadingsien pituus

    if docHeadingsLength != tocHeadingsLength:
        return False
        # errors.append('Sisällysluettelon otsikkoja on eri määrä kuin dokumentin otsikkoja.')
        #------------------------ # if-else shorthand: x = z if condition else y
        # if docHeadingsLength > tocHeadingsLength: listLength = tocHeadingsLength
        #--------------------------------- else: listLength = docHeadingsLength

    i = 0
    while i < listLength:
#         if string.find (tocHeadings[i], docHeadings[i]) == -1:
        if tocHeadings[i].find(docHeadings[i]) == -1:
            return False
        i += 1

    bookmarkStarts = _getElementsWithinElement(docXml, 'w:bookmarkStart')
    instrElements = _getElementsWithinElement(docXml, 'w:instrText')
    for element in docTocStyles:
        instrElements = _getElementsWithinElement(element, 'w:instrText')
        for instrElement in instrElements:
            instrElementValue = instrElement.firstChild.nodeValue
            if instrElementValue.find('PAGEREF') != -1: # We only want to handle the tags with value including PAGEREF.
                for bookmark in bookmarkStarts:
                    bookmarkNameValue = bookmark.getAttribute('w:name')
                    if bookmarkNameValue.find(instrElementValue) != -1: # if the same code is in bookmarkStart
                        return False

    return True

def checkTOC(document):
    ''' Check if table of contents is done correctly. It has to have a page break before (and after) it.

    @see: checkTocContent -- calls the method if there's a table of contents to be found.

    @note: XML example:

    <w:p w:rsidR="004A16ED" w:rsidRDefault="004A16ED" w:rsidP="006158B0">

    <w:pPr>


    <w:pStyle w:val="Otsikko"/>

    </w:pPr>

    <w:r w:rsidRPr="006158B0">

    <w:lastRenderedPageBreak/>

    <w:t>SISALLYSLUETTELO</w:t>

    </w:r>

    </w:p>

    <w:p w:rsidR="002274FC" w:rsidRDefault="00FA6E61">

    <w:pPr>

    <w:pStyle w:val="Sisluet1"/>

    @return: True if toc is made correctly, False otherwise.
    '''

    styleId = _getStyleIdByName("toc 1", document['word/styles.xml']) # Sisluet1
    #if (styleId == None): errors.append('There is no table of contents.')
    pStyles = _getElementsWithinElement(document['word/document.xml'], 'w:pStyle')

    if pStyles is None:
        return False
```

```
    for style in pStyles:
        if (style.getAttribute('w:val') == styleId):
            return True

            #checkTocContent() # tarkistaa onko sisällysluettelo päivitetty


#            try: #TODO: should a page break after the table of contents as well, what about section break?
#                #TODO: it's about section break, this code was made for page break.
#                if (pageBreak == style.parentNode.parentNode.previousSibling.getElementsByTagName('w:r')[0].firstChild.tagN
#                    print "There is a page break before table of contents."
#                    return
#            except IndexError:
#                errors.append("There is no page break before the table of contents.")
#                return

    return False
    #if not tocExists: errors.append("There is no table of contents at all.")


    #return styleId

def checkCoverPage(document):
    ''' Checks if the front page is done correctly

    @return: coverPageText dictionary containing True or False values.
    '''

    # if rakenne on oikein, do -- rakenteen tarkastus lisättävä alkuun

    coverPageText = { 'email': False,
                      'name': False,
                      'title': False }

    docXml = document['word/document.xml']
    coreXml = document['docProps/core.xml']

    paragraphs = _getElementsWithinElement(docXml, 'w:p')
    allSectionProperties = getSectionElementsBySections(docXml, 0)

    firstPageText = ''
    lastParagraphOfFirstPage = allSectionProperties[-1].parentNode.parentNode

    # Saves the content of the first page, getting text from the beginning and
    # breaks the loop when the sectPr-node appears.
    # Pitää katsoa, ettei etusivulla ole sisällysluetteloa ym., koska jos section-breakit on väärin,
    # "etusivun" tietoihin voi valua sisällysluettelo.
    for element in paragraphs:
        firstPageText += getTextContent(element)
        if element.isSameNode(lastParagraphOfFirstPage):
            break

    # title pitää myös löytyä täältä
    if firstPageText.find('@') != -1: # sähköpostiosoite tulee lopuksi käyttöliittymästä
        coverPageText['email'] = True
    if firstPageText.find(_getLastModifier(coreXml)) != -1:
        coverPageText['name'] = True
    title = _getTitle(coreXml)
    if title is not None:
        if firstPageText.find(title) != -1:
            coverPageText['title'] = True

    return coverPageText

def getRelsTargetByRId(rId, rels):
    '''Returns the value of Target attribute of a Relationship element with the given id in a given rels file.
    The value of Target attribute can be for example a relative path to local XML files or images. It can also be a hyperlin

    @param rId: Id attribute value of a Relationship element.
    @param rels: rels file as a DOM tree.

    @return: The value of Target attribute if found.
    '''
    for relationship in rels.getElementsByTagName('Relationship'):
        if rId == relationship.getAttribute('Id'):
            return relationship.getAttribute('Target')

def getParentParagraph(element, tag='w:p'):
    ''' Returns the parent <w:p>-element of a given element if there is one.

    @param element: The element whose parent <w:p> element is searched for.
    @param tag: The parent tagname, defaults to 'w:p'.

    @return: The parent element, or None if no parent is found.
    '''
```

```
    parent = element.parentNode

    while parent is not None:
        try:
            if parent.tagName == tag:
                return parent
            else:
                parent = parent.parentNode
        except AttributeError:
            return None
    return None

def checkImages(document):
    ''' Check if there is an image in the document.

    @return: True if even one image is found, False otherwise.
    '''

    #TODO: what is the difference between pic:pic and w:pict?
    #w:pict is used when pasting a chart from powerpoint or excel?
    picElements = document['word/document.xml'].getElementsByTagName('pic:pic')
    pictElements = document['word/document.xml'].getElementsByTagName('w:pict')

    if len(picElements) > 0:
        return True
    if len(pictElements) > 0:
        return True
    return False

def getImagePaths(document):
    ''' Gets the image paths or the file names of the images used in the document.

    @return: The image targets as strings in a list.
    '''
    targets = []

    picElements = document['word/document.xml'].getElementsByTagName('pic:pic')
    picElements += document['word/document.xml'].getElementsByTagName('w:pict')

    for picElement in picElements:
        picRId = picElement.getElementsByTagName('a:blip')[0].getAttribute('r:embed')
        targets.append(getRelsTargetByRId(picRId, document['word/_rels/document.xml.rels']))
    return targets

def checkImageCaptions(document):
    '''Checks if the next paragraph after a picture paragraph uses the caption style.

    Also checks that the caption contains an automatic field.
    Goes through all picture paragraphs.

    @return: True if all images have captions, False otherwise.
    '''
#    <w:p w:rsidR="0011423F" w:rsidRDefault="00FE23CD" w:rsidP="00A72640">
#    <w:pPr>
#        <w:pStyle w:val="Kuvanotsikko"/>
#    </w:pPr>
#    <w:bookmarkStart w:id="10" w:name="_Ref247712443"/>
#    <w:r>
#        <w:t xml:space="preserve">Kuva </w:t>
#    </w:r>
#    <w:fldSimple w:instr=" SEQ Kuva \* ARABIC ">
#        <w:r>
#            <w:t>1</w:t>
#        </w:r>
#    </w:fldSimple>
#    -
#    <w:r w:rsidR="00876DBA">
#        <w:t>Kurssijako</w:t>
#    </w:r>
#    <w:bookmarkEnd w:id="10"/>
#    </w:p>
#
    styleXml = document['word/styles.xml']

    picParagraphs = []
    picElements = document['word/document.xml'].getElementsByTagName('pic:pic')
    picElements += document['word/document.xml'].getElementsByTagName('w:pict')

    for pic in picElements:
        picParagraphs.append(getParentParagraph(pic))

    if picParagraphs is None:
        return False

    for p in picParagraphs:
        try:
```

```
            captionParagraph = p.nextSibling
            captionParagraphPpr = captionParagraph.getElementsByTagName('w:pPr')[0]
            captionParagraphStyleID = captionParagraphPpr.getElementsByTagName('w:pStyle')[0].getAttribute('w:val')
            captionParagraphStyleName = _getStyleNameById(captionParagraphStyleID, styleXml)
            if captionParagraphStyleName != "caption":
                return False
        except:
            return False

        #Find 'SEQ' in either one of the attribute values in the paragraph or as a text node. It indicates an automatic fiel
        if getAttributeContent(captionParagraph).find('SEQ') == -1 and \
           getTextContent(captionParagraph).find('SEQ') == -1:
            return False

    return True

def checkStyleUsage(document, errorIdsAndPositions):
    '''Checks that text paragraphs are using styles and that no manual style definitions are made.

    Goes through all paragraph-elements in a document looking for <w:pStyle>-elements.
    Gets the style definitions to see if there are manual changes.

    @note: Exception:

    Automatically generated table on contents can contain "manual" style definitions.
    The <w:sectPr> elements within paragraph elements are skipped also.

    @param errorIdsAndPositions: A dict for error strings. Should contain keys 'manualChanges' and 'styleNotUsed'.

    @return: True if nothing was found, False if even one error was found.
    '''

    paragraphs = document['word/document.xml'].getElementsByTagName('w:p')
    styleXml = document['word/styles.xml']

    for p in paragraphs:
        styleDefinitions = {'w:autoRedefine': None,
                        'w:left': None,
                        'w:right': None,
                        'w:firstLine': None,
                        'w:line': None,
                        'w:before': None,
                        'w:after': None,
                        'w:widowControl': None,
                        'w:jc': None,
                        'w:sz': None,
                        'w:ascii': None,
                        'w:asciiTheme': None,
                        'w:b': None,
                        'w:i': None,
                        'w:u': None,
                        #'w:outline': None,
                        #'w:numId': None,
                        #'w:ilvl': None,
                        #'w:lang': None,
                        'w:keepNext': None,
                        'w:keepLines': None,
                        'w:pageBreakBefore': None}

        try:
            style = p.getElementsByTagName('w:pStyle')[0].getAttribute('w:val')
            style = _getStyleNameById(style, styleXml)

# It seems that Word (2010) makes style definitions in document.xml when generating an automatic table of contents:
            if style.startswith('toc'):
                continue
#            if style != "":
#                print style

# TODO: check the empty paragraphs content to prevent false positives, are they empty text paragraphs or maybe a picture par
# Now just leaves the paragraphs with no text be and doesn't give an error.

        except:
            #pContent = getTextContent(p)
            pContent = _getTextFromParagraph(p)
            if pContent.strip() != "":
                #errors.append("No style used in paragraph: " + str(pContent[:25]))
                errorIdsAndPositions['styleNotUsed'].append(pContent[:30])

        try:
            for paragraphProperties in p.getElementsByTagName('w:pPr'):
                for propertyElement in paragraphProperties.childNodes:
# We don't want section properties to be mixed up as manually made style definitions.
                    if propertyElement.tagName != "w:sectPr":
                        styleDefinitions = _getStyleDefinitions(propertyElement, styleDefinitions)
            for runProperties in p.getElementsByTagName('w:rPr'):
```

```
                styleDefinitions = _getStyleDefinitions(runProperties, styleDefinitions)

            for key in styleDefinitions.keys():
                if styleDefinitions[key] is not None:
                    pContent = _getTextFromParagraph(p)
                    if pContent.strip() != "":
                        errorIdsAndPositions['manualChanges'].append(pContent[:50])
                        break
        except:
            continue

    for key in errorIdsAndPositions.keys():
        if len(errorIdsAndPositions[key]) > 0:
            return False
    return True

def checkEndnotesAndFootnotes(document):
    ''' Checks if there is an endnote or a footnote in the document.

    Looks for w:endnoteReference and w:footnoteReference elements.

    @return: True if an endnote or a footnote is found, False otherwise.
    '''
    docXml = document["word/document.xml"]

    endnotes = docXml.getElementsByTagName('w:endnoteReference')
    footnotes = docXml.getElementsByTagName('w:footnoteReference')

    if len(endnotes) != 0: return True
    if len(footnotes) != 0: return True

    # TODO: doublecheck: find the endnote with the id in endnotes.xml. That's where the endnote text is located.
    # TODO: doublecheck: find the footnote in footnotes.xml

    return False

def checkCrossRefrenceToImageCaption(document):
    ''' Goes through images' captions looking for a reference. Then checks if the caption is referenced somewhere.

    @return: True if a cross reference is found, False otherwise.
    '''
    #TODO: not implemented in word_processing 13.5.2011

    docXml = document['word/document.xml']

    picParagraphs = []
    picElements = document['word/document.xml'].getElementsByTagName('pic:pic')
    picElements += document['word/document.xml'].getElementsByTagName('w:pict')

    for pic in picElements:
        picParagraphs.append(getParentParagraph(pic))

    for p in picParagraphs:
        captionParagraph = p.nextSibling
        try:
            bookmarkStartElement = captionParagraph.getElementsByTagName('w:bookmarkStart')[0]
        except:
            # raise an error: errors.append('Picture\'s caption reference not found.')
            return False
        reference = bookmarkStartElement.getAttribute('w:name')

        for element in docXml.getElementsByTagName('w:instrText'):
            elementText = getTextContent(element)
            if elementText.find(reference) != -1:
                return True
    # raise an error: errors.append('Reference to picture caption not found.')
    return False

def _getElementByAttributeValue(nodeList, attributeName, attributeValue):
    '''Gets an element by an attribute value.

    @param nodeList: A list of elements to be searched for.
    @param attributeName: The name of the wanted attribute.
    @param attributeValue: The wanted value of the attribute.

    @return: The element, if it has an attribute with the wanted value, None otherwise.
    '''
    for element in nodeList:
        if element.getAttribute(attributeName) == attributeValue:
            return element
    return None

def _isStyleUsed(document, styleName):
    '''Checks that a style is used in the document.

    @param styleName: The name of the style looked for.
```

```
    @return: True if the style is used, False otherwise.
    '''
    docXml = document['word/document.xml']
    styleXml = document['word/styles.xml']

    styleId = _getStyleIdByName(styleName, styleXml)
    bodyParagraphs = _getParagraphElementsByStyleId(docXml, styleId)

    if len(bodyParagraphs) > 0:
        return True
    return False

#def checkHeadingUsage(document):
#    '''Check if heading styles are used in the document.
#
#    @return: True if heading styles are used, False otherwise.
#    '''
#    docXml = document['word/document.xml']
#    styleXml = document['word/styles.xml']
#
#    headingParagraphs = _getParagraphElementsBySequentialStyleName("heading", styleXml, docXml)
#    if len(headingParagraphs) == 0:
#        #errors.append("No heading styles used in this document!")
#        return False
#
#    usedHeadingStyles = []
#    for heading in headingParagraphs:
#        styleId = heading.getElementsByTagName('w:pStyle')[0].getAttribute('w:val')
#        if usedHeadingStyles.count(styleId) == 0:
#            usedHeadingStyles.append(str(styleId))
#
#    #FIXME: not the most dynamic way:
#    if len(usedHeadingStyles) < 2:
#        return False
#
#    return True

def checkHeadingNumbering(document, errorIdsAndPositions):
    '''Checks the headings in the document.

    Goes through the heading styles used in the document checking that they use a multilevel numbering,
    the numbering is done correctly using styles and that the numbering is connected to other heading styles.

    Gets all the heading styles used in the document.
    Searches for the heading's numbering definition reference in styles.xml.
    Next searches the associated numbering definition in numbering.xml.
    Next searches the correct numbering level definition associated to the heading.
    Checks that the numbering is multilevel and done correctly using the heading styles.

    @note: XML example:

    styles.xml:

    <w:style w:type="paragraph" w:styleId="Heading2"> - Heading 2 style definition

    <w:name w:val="heading 2"/>

    <w:pPr>

    <w:numPr>

    <w:ilvl w:val="1"/> - Numbering Level Reference

    <w:numId w:val="1"/> - Numbering Definition Instance Reference

    </w:numPr>

    <w:outlineLvl w:val="1"/>

    </w:pPr>

    </w:style>

    numbering.xml:

    <w:abstractNum w:abstractNumId="0"> - Abstract Numbering Definition

    <w:multiLevelType w:val="multilevel"/> - Abstract Numbering Definition Type

    <w:lvl w:ilvl="0"> - </w:lvl> - Numbering Level Definition

    <w:lvl w:ilvl="1"> - Numbering Level Definition

    <w:start w:val="1"/> - Starting Value
```

```
<w:numFmt w:val="decimal"/> - Numbering Format

<w:pStyle w:val="Heading2"/> - Paragraph Style's Associated Numbering Level

<w:lvlText w:val="%1.%2"/> - Numbering Level Text

<w:lvlJc w:val="left"/> - Justification

<w:pPr> - Numbering Level Associated Paragraph Properties

<w:ind w:left="576" w:hanging="576"/>

</w:pPr>

</w:lvl>

</w:abstractNum>

<w:num w:numId="1"> - Numbering Definition Instance

<w:abstractNumId w:val="0"/> - Abstract Numbering Definition Reference

</w:num>

@param errorIdsAndPositions: A dict for appending errors in key - stringlist pairs.
Should contain the following keys:
  - 'manualNumbering' -- numbering is done manually somehow.
  - 'styleNotUsed' -- an expected heading style is not used.
  - 'differentNumbering' -- some heading style is using different numbering than some other heading styles.
  - 'notMultilevel' -- the numbering is not multilevel.
  - 'outlineLvl' -- the outline of a heading style is not correct.
  - 'numStart' -- the numbering doesn't start at 1.
  - 'numWrong' -- the numbering is somehow not done with styles.
  - 'numFormat' -- the numbering format is not correct.
  - 'notSequential' -- heading styles are not used correctly in a row for example heading 3 is used after heading 1.
'''

#errorIdsAndPositions = {'manualNumbering': None}
docXml = document["word/document.xml"]
styleXml = document["word/styles.xml"]

try:
    numXml = document['word/numbering.xml']
    #numFile = zip.read('word/numbering.xml')
    #numXml = xml.dom.minidom.parseString(numFile)
except:
    #errors.append("No heading numbering used at all.")
    return False


headingParagraphs = _getParagraphElementsBySequentialStyleName("heading", styleXml, docXml)
#@see: checkHeadingUsage
#if len(headingParagraphs) == 0:
#    errors.append("No heading styles used in this document!")
#    return

usedHeadingsStyleIds = []
previousHeadingLevel = 0
for heading in headingParagraphs:
    styleId = heading.getElementsByTagName('w:pStyle')[0].getAttribute('w:val')
    if len(heading.getElementsByTagName('w:ilvl')) > 0 or \
       len(heading.getElementsByTagName('w:numId')) > 0:
        errorIdsAndPositions['manualNumbering'] = getTextContent(heading)
        # errors.append("Manual numbering definitions made in heading: " + getTextContent(heading))
    if usedHeadingsStyleIds.count(styleId) == 0:
        usedHeadingsStyleIds.append(str(styleId))
    headingLevel = int(styleId[len(styleId) - 1])
    if fabs(headingLevel - previousHeadingLevel) > 1:
        #errors.append("Otsikoita ei ole käytetty oikealla tavalla peräkkäin.")
        errorIdsAndPositions['notSequential'] = getTextContent(heading)
    previousHeadingLevel = headingLevel

#Sort the list: ['Heading1', 'Heading2', 'Heading3', ...]
usedHeadingsStyleIds.sort(cmp=None, key=None, reverse=False)
#print usedHeadingsStyleIds

previousNumId = None

for headingStyleId in usedHeadingsStyleIds:

    headingLevel = int(headingStyleId[len(usedHeadingsStyleIds[0]) - 1])

    headingStyleElement = _getStyleElementById(headingStyleId, styleXml)

    # Get the numbering definitions of the heading style.
    # Default ilvl value to 0 -> ilvl-element not found (level is 0).
```

```
        styleDefinitions = {'w:ilvl': '0', 'w:numId': None, 'w:outlineLvl': None}
        styleDefinitions = _getStyleDefinitions(headingStyleElement, styleDefinitions)

        # Chekc that the numbering style definitions are OK.
        if styleDefinitions['w:numId'] is None:
            errorIdsAndPositions['styleNotUsed'] = headingStyleId
            #errors.append(headingStyleId + " numbering is not used.")
            #return
        if previousNumId is not None and styleDefinitions['w:numId'] != previousNumId:
            errorIdsAndPositions['differentNumbering'] = headingStyleId
            #errors.append(headingStyleId + " is using different numbering as the previous level heading style.")
        previousNumId = styleDefinitions['w:numId']
        if int(styleDefinitions['w:ilvl']) != headingLevel - 1:
            errorIdsAndPositions['notMultilevel'] = headingStyleId
            #errors.append(headingStyleId + " numbering level is not correct, numbering is not multilevel.")
        if int(styleDefinitions['w:outlineLvl']) != headingLevel - 1:
            errorIdsAndPositions['outlineLvl'] = headingStyleId
            #errors.append(headingStyleId + " outline level is not correct.")

        # Find the numbering definition element associated to the heading style.
        # Get the abstract numbering definition id from the numbering definition element.
        # Find the abstract numbering definition element with the correct id.
        # Find the numbering level definition with the same level that the heading style.
        try:
            numElement = _getElementByAttributeValue(numXml.getElementsByTagName('w:num'), 'w:numId', styleDefinitions['w:nu
            abstractNumId = numElement.getElementsByTagName('w:abstractNumId')[0].getAttribute('w:val')
            absNumElement = _getElementByAttributeValue(numXml.getElementsByTagName('w:abstractNum'), 'w:abstractNumId', abs
            lvlElement = _getElementByAttributeValue(absNumElement.getElementsByTagName('w:lvl'), 'w:ilvl', styleDefinitions
        except:
#            errors.append(headingName + " numbering level definitions not found.")
            continue

        # Get the numbering level definitions.
        numDefinitions = {'w:start': None, 'w:numFmt': None, 'w:pStyle': None, 'w:lvlText': None, 'w:lvlJc': None, 'w:tentat
        numDefinitions = _getStyleDefinitions(lvlElement, numDefinitions)

# TODO: should we check that the numbering is in format 1, 1.1, 1.1.1 etc ?
        if numDefinitions['w:start'] != '1':
            errorIdsAndPositions['numStart'] = headingStyleId
            #" numbering doesn't start at number 1.")
        if numDefinitions['w:pStyle'] != headingStyleId:
            errorIdsAndPositions['numWrong'] = headingStyleId
            #" numbering is not done correctly using heading styles.")
        if numDefinitions['w:numFmt'] != "decimal":
            errorIdsAndPositions['numFormat'] = headingStyleId
            #errors.append(headingStyleId + " numbering format is not a decimal number.")

    return True

def _getParagraphElementsByStyleId(docXml, styleId):
    ''' Gets all paragraph-elements in the document by a style id.'''
    paragraphList = []

    for p in docXml.getElementsByTagName('w:p'):
        try:
            if styleId == p.getElementsByTagName('w:pStyle')[0].getAttribute('w:val'):
                paragraphList.append(p)
        except:
            continue
            #errors.append("No style used in paragraph: " + getTextContent(p))
    return paragraphList

def _getParagraphElementsBySequentialStyleName(styleNamePrefix, styleXml, docXml):
    ''' Return all paragraph elements that use a style name with a sequential numbering.

    Gets all paragraphs that use styles with stylenames for example heading 1, heading 2, etc or
    index 1, index 2, etc.

    @param styleNamePrefix: The prefix of the sequential style name.
    '''
    paragraphs = []
    i = 1
    styleNamePrefix = styleNamePrefix.strip() + " "

    while(True):
        styleId = _getStyleIdByName(styleNamePrefix + str(i), styleXml)
        if styleId is None:
            break
        else:
            paragraphs += _getParagraphElementsByStyleId(docXml, styleId)
            i += 1
    return paragraphs

def checkIndex(document):
    '''Checks that the document has an automatically made index.
```

```
    @return: False if an index is missing, '2' if index is not automatically made and True if everything was OK.
    '''
    docXml = document['word/document.xml']
    styleXml = document['word/styles.xml']

    indexParagraphs = _getParagraphElementsBySequentialStyleName("index ", styleXml, docXml)
    if len(indexParagraphs) == 0:
        return False
    #if len(indexParagraphs) != 0: return True

    # The previous w:p element of the first index entry should be something like this:
    #<w:p w:rsidR="002F2A09" w:rsidRDefault="00CA51D5">
        #<w:pPr>
            #<w:sectPr w:rsidR="002F2A09" w:rsidSect="002F2A09"> --- </w:sectPr>
        #</w:pPr>
        #<w:r>
            #<w:fldChar w:fldCharType="begin"/>
        #</w:r>
        #<w:r>
            #<w:instrText xml:space="preserve"> INDEX \c "2" \z "1035" </w:instrText>
        #</w:r>
        #<w:r>
            #<w:fldChar w:fldCharType="separate"/>
        #</w:r>
    #</w:p>

    # Search the 'instrText' field element and be sure that it's an index field element.
    try:
        indexFieldCodeElement = indexParagraphs[0].previousSibling.getElementsByTagName('w:instrText')[0]
    except:
        indexFieldCodeElement = None
    # There can be a section brake between the first index entry and the field declaration?
    if indexFieldCodeElement is None:
        try:
            indexFieldCodeElement = indexParagraphs[0].previousSibling.previousSibling.getElementsByTagName('w:instrText')[0
        except:
            indexFieldCodeElement = None

    if indexFieldCodeElement is None:
        #errors.append('Index is not a field - make the index automatically, not manually.')
        return '2'
    elif getTextContent(indexFieldCodeElement).find('INDEX') == -1:
        #errors.append('INDEX-text not found in field declaration - make the index automatically, not manually.')
        return '2'

    return True

def checkIndexContent(document):
    ''' Checks that the document has a index that is not empty, and that the index entries are referenced somewhere in the d

    First gets all the index styles' definitions from styles.xml and finds paragraphs using the styles in the document.xml.
    Checks that there is a field code element indicating that the index is generated automatically.
    Collects the content of the index and checks it isn't empty.
    Finds references to the index entries and matches them to the index content.

    @note: XML example:

    Index example:

    <w:p w:rsidR="002F2A09" w:rsidRDefault="00CA51D5">

    <w:r>

    <w:fldChar w:fldCharType="begin"/>

    </w:r>

    <w:r>

    <w:instrText xml:space="preserve"> INDEX \c "2" \z "1035" </w:instrText>

    </w:r>

    <w:r>

    <w:fldChar w:fldCharType="separate"/>

    </w:r>

    </w:p>

    <w:p w:rsidR="002F2A09" w:rsidRDefault="002F2A09">

    <w:pPr>

    <w:pStyle w:val="Index1"/>
```

```
<w:tabs>

<w:tab w:val="right" w:leader="dot" w:pos="3950"/>

</w:tabs>

</w:pPr>

<w:r>

<w:t>Index entry level 1</w:t>

</w:r>

</w:p>

Reference example:

<w:r w:rsidR="00B27B47">

<w:instrText xml:space="preserve"> XE "</w:instrText>

</w:r>

<w:r w:rsidR="00B27B47" w:rsidRPr="00B27B47">

<w:instrText>Level 1 entry</w:instrText>

</w:r>

<w:r w:rsidR="00B27B47" w:rsidRPr="00B27B47">

<w:instrText>:</w:instrText>

</w:r>

<w:r w:rsidR="00B27B47" w:rsidRPr="0011587C">

<w:instrText>Level 2 entry</w:instrText>

</w:r>

@return: '3' if the index is empty, '4' if the content does not match with the document and True if everything went OK.
'''
docXml = document['word/document.xml']
styleXml = document['word/styles.xml']

indexParagraphs = _getParagraphElementsBySequentialStyleName("index ", styleXml, docXml)
indexTextContent = dict()
for p in indexParagraphs:
    textContent = getTextContent(p)
    if textContent is not None and textContent != "":
        indexTextContent[textContent] = None
if len(indexTextContent) == 0:
    #errors.append('Index is empty.')
    return '3'

documentFieldTexts = ""
for pElement in docXml.getElementsByTagName('w:instrText'):
    documentFieldTexts += getTextContent(pElement)

# Index entry reference example: 'XE "MainEntry"', 'XE "MainEntry:SubEntry"' or even 'XE "MainEntry:Heading" "Subentry:H
# Check that the entries are actually included in the index:
# Parse the string containing all text content of the w:instrText-elements.
# First split at XE_ (where _ is whitespace), next split at \" and finally split at \:.
# Compare the final index entry candidate to the index entries and visa versa to see if they match.
# If finally some index entry doesn't have any matches, the entry is probably made manually. It's referenced nowhere!
# TODO: check that the reference makes actually sense? Page number to the index comes from the page where the reference

indexReferenceFieldsContent = []
for field in documentFieldTexts.split('XE '):
    for candidate in field.split('\"'):
        for finalCandidate in candidate.split(":"):
            if finalCandidate.strip() != "":
                indexReferenceFieldsContent.append(finalCandidate)


for indexReferenceComponent in indexReferenceFieldsContent:
    #print indexReferenceComponent
    for key in indexTextContent.keys():
        if key.find(indexReferenceComponent) != -1:
            indexTextContent[key] = True
            break
        elif indexReferenceComponent.find(key) != -1:
            indexTextContent[key] = True
```

```
                break

    for key in indexTextContent.keys():
        #print key + " - " + str(indexTextContent[key])
        if indexTextContent[key] is None:
            #errors.append('No references found for index entry ' + key + '.')
            return '4'
    return True

def checkDoubleWhitespaces(document):
    '''Checks double whitespaces in the document.

    @return: The amount of occurrences of the double whitespaces found in the document, False otherwise.
    '''
    return checkStringFromDocument(document['word/document.xml'], '  ')

def checkAsterisk(document):
    '''Checks the *-character in the document.

    @return: The amount of occurrences of the asterisks found in the document, False otherwise.
    '''
    return checkStringFromDocument(document['word/document.xml'], '*')

def checkStringFromDocument(docXml, string):
    '''Checks if a string is found in the text content of the document (in the w:t-elements).
    If string is found, returns how many occurences were found in a paragraph.

    @return: The amount of occurrences of the string is found in the document, False otherwise.
    '''
    found = False
    count = 0
    for p in docXml.getElementsByTagName('w:p'):
        textContent = ""
        for textElement in p.getElementsByTagName('w:t'):
            textContent += getTextContent(textElement)
        occurrences = textContent.count(string)
        if occurrences > 0:
            count += occurrences
#            errors.append("\"" + string + "\" occurs " + str(occurrences) + " time(s) in paragraph: " + textContent[:25])
            found = True
    if found is True:
        return count
    return found
#    for p in docXml.getElementsByTagName('w:p'):
#        if(checkStringFromContent(p, "  ")):
#            errors.append("Double whitespace in paragraph: " + getTextContent(p))

def checkTabs(document):
    '''Checks if the tabulator is used in the document.

    @note: Exceptions:

      - automatically generated table of contents and index contain tabulators.

      - before an automatically generated index there is a paragraph-element with <instrText>-element and a <tab>-element.

    @return: The amount of the tabulator occurrences found in the document, False if none was found.
    '''
    #TODO: More exceptions?
    styleXml = document['word/styles.xml']
    tabParagraphs = document['word/document.xml'].getElementsByTagName('w:tab')
    tabCount = 0

    if len(tabParagraphs) == 0:
        return False

    #tabParagraphContent = []
    #tabParagraphContent = dict()

    for tab in tabParagraphs:
        tabParent = getParentParagraph(tab, 'w:p')
        try:
            tabParentStyleId = tabParent.getElementsByTagName('w:pStyle')[0].getAttribute('w:val')
        except:
            continue
        if _getStyleNameById(tabParentStyleId, styleXml).startswith('toc') or \
           _getStyleNameById(tabParentStyleId, styleXml).startswith('index'):
            continue
        else:
            if getTextContent(tabParent).find('INDEX') != -1:
                continue
            else:

                tabCount += 1
#            print getTextContent(tabParent)
            #tabParagraphContent.append(getTextContent(tabParent))
```

```
#            try:
#                tabParagraphContent[getTextContent(tabParent)] += 1
#            except KeyError:
#                tabParagraphContent[getTextContent(tabParent)] = 1
#
#    if len(tabParagraphContent) == 0:
    if tabCount == 0:
        return False

    return tabCount

def isParagraphEmpty(p, styleXml):
    '''Checks if a paragraph is empty.

    @note: Expections:

    Picture in the document produces an empty paragraph.
    Empty table cell produces an empty paragraph.
    A table produces an empty paragraph right after the table.
    Objects and graphics produce an empty paragraph.
    ...

    @param p: The paragraph element under inspection.

    @return: False if the paragraph is not empty, True if it is empty.
    '''
    #FIXME: these are surely not the only exceptions. Add more exceptions.
    pContent = _getTextFromParagraph(p).strip()
    if len(pContent) == 0:
        if len(p.getElementsByTagName('pic:pic')) > 0:
            return False
        if len(p.getElementsByTagName('w:sectPr')) > 0:
            return False
        if len(p.getElementsByTagName('w:pict')) > 0:
            return False
        if len(p.getElementsByTagName('w:object')) > 0:
            return False
        if len(p.getElementsByTagName('a:graphic')) > 0:
            return False
        if getParentParagraph(p, 'w:tbl') is not None:
            return False

        #TODO: try-except on previousSiblings
        if p.previousSibling is not None:
            if p.previousSibling.tagName == 'w:tbl':
                return False
            styleId = _getParagraphStyleId(p.previousSibling)
            if styleId is not None:
                styleName = _getStyleNameById(styleId, styleXml)
                if styleName is not None:
                    if styleName.find('toc') != -1:
                        return False
        if p.previousSibling is not None:
            styleId = _getParagraphStyleId(p.previousSibling)
            if styleId is not None:
                styleName = _getStyleNameById(styleId, styleXml)
                if styleName is not None:
                    if styleName.find('index') != -1:
                        return False
            if p.previousSibling.previousSibling is not None:
                styleId = _getParagraphStyleId(p.previousSibling.previousSibling)
                if styleId is not None:
                    styleName = _getStyleNameById(styleId, styleXml)
                    if styleName is not None:
                        if styleName.find('index') != -1:
                            return False

        #print getTextContent(p)
        #print getTextContent(p.previousSibling)
        #print getTextContent(p.previousSibling.previousSibling)
        return True
    return False

def checkEmptyParagraphs(document):
    ''' Finds all empty paragraphs in the document.

    @note: Expections:

    Picture in the document produces an empty paragraph.
    Empty table cell produces an empty paragraph.
    A table produces an empty paragraph right after the table.
    ...?

    @return: amount of empty paragraph occurrences in the document, False if none was found.
    '''
    paragraphs = document['word/document.xml'].getElementsByTagName('w:p')
```

```
    #emptyParagraphs = dict()
    emptyParagraphsCount = 0

    for p in paragraphs:
        result = isParagraphEmpty(p, document['word/styles.xml'])

        if result is True:
            emptyParagraphsCount += 1

    if emptyParagraphsCount == 0:
        return False
    else:
        return emptyParagraphsCount

def checkList(document, listName='List'):
    ''' Goes through all paragraph elements in the document looking for paragraphs that use some list style.

    @param listName: The list stylename we want to check. Defaults to 'List',
                     which finds list styles such as 'List', 'List Bullet', 'List Numbered'.

    @return: True, if a list style is used in the document, False otherwise.
    '''
    docXml = document['word/document.xml']
    styleXml = document['word/styles.xml']

    for p in docXml.getElementsByTagName('w:p'):
        styleId = _getParagraphStyleId(p)
        if styleId is not None:
            styleName = _getStyleNameById(styleId, styleXml)
            if styleName.find(listName) != -1:
                return True

    return False

def checkSpreadsheetChart(document):
    '''Checks that the document has a chart copied from a spreadsheet document.
    The Chart must be pasted as a link.
    '''

    #TODO: not implemented in word_processing 13.5.2011
    docXml = document['word/document.xml']
    docRelsXml = document['word/_rels/document.xml.rels']

    objectElements = docXml.getElementsByTagName('w:object')
    if len(objectElements) == 0:
        return False

    for objectElement in objectElements:
        if len(objectElement.getElementsByTagName('v:formulas')) > 0:

            try:
                OLEObjectElement = objectElement.getElementsByTagName('o:OLEObject')[0]
            except:
                continue

            if OLEObjectElement.getAttribute('ProgID').find('Excel') != -1:
                #print "Spreadsheet chart is not from Excel."
                #print OLEObjectElement.getAttribute('Type')
                if OLEObjectElement.getAttribute('Type') == 'Link':

                    rid = OLEObjectElement.getAttribute('r:id')
                    target = getRelsTargetByRId(rid, docRelsXml)
                    targetChart = target.split("!")
                    targetChart.reverse()
                    #Example: targetChart[0] = %5bmalli.xlsx%5dmalli%20Chart%201
                    targetChartName = targetChart[0]

                    #TODO: more effective examination required, just check that there is more than three %-characters.
                    if targetChartName.count('%') < 3:
                        return False

                    #print targetChartName


                    return True
                else:
                    return "Spreadsheet object is not pasted as a link."
    return False

def checkSpreadsheetTable(document):
    '''Checks that the document has a table copied from a spreadsheet document.
    For now checks that the table is pasted as a link.
    '''
    docXml = document['word/document.xml']
    docRelsXml = document['word/_rels/document.xml.rels']
```

```
    objectElements = docXml.getElementsByTagName('w:object')
    if len(objectElements) == 0:
        return False

    for objectElement in objectElements:
        if len(objectElement.getElementsByTagName('v:formulas')) == 0:

            try:
                OLEObjectElement = objectElement.getElementsByTagName('o:OLEObject')[0]
            except:
                continue

            if OLEObjectElement.getAttribute('ProgID').find('Excel') != -1:
                #print "Spreadsheet chart is not from Excel."
                #print OLEObjectElement.getAttribute('Type')
                if OLEObjectElement.getAttribute('Type') == 'Link':
                    rid = OLEObjectElement.getAttribute('r:id')
                    target = getRelsTargetByRId(rid, docRelsXml)
                    targetChart = target.split("!")
                    targetChart.reverse()
                    targetTableCells = targetChart[0]

                    #Example: targetTableCells = R1C1:R7C4
                    #TODO: more effective examination might be required, just check that 2 R- and 2 C-characters are found.
                    if targetTableCells.count('R') != 2:
                        return False
                    if targetTableCells.count('C') != 2:
                        return False
                    #print targetTableCells

                    return True
                else:
                    return "Spreadsheet object is not pasted as a link."
    return False

def checkPresentationGraphicsChart(document):
    '''Checks that the document contains a chart pasted from PowerPoint as a vector graphics picture or as an object.
    Doesn't really know if the picture or object is actually from PowerPoint!
    '''

    #TODO: not implemented in word_processing 13.5.2011

    docRelsXml = document['word/_rels/document.xml.rels']

    pictureTargets = getImagePaths(docRelsXml)
    for target in pictureTargets:
        if target.endswith('.emf') is True:
            # .emf vector graphics picture was found.
            return True
        if target.endswith('wmf') is True:
            return True


    # TODO: check the object if no vector graphics picture is found.

    # Normal jpg or png pictures may be inside the w:drawing-element.
    #drawingElements = doc.getElementsByTagName('w:drawing')
    #if len(drawingElements) > 0:
    #    return True

    return False
```