Project Plan

coala offers a unified command line interface for detecting and fixing code with the help of static analysis routines in the form of bears. The fixes provided by the different bears are converted to a Diff class instance which coala later uses to display the fix along with the analysis results to the users and lets the users choose the appropriate actions.

Binary Diff

Currently coala supports unified diff handling which produces a smaller diff with old and new text presented immediately adjacent. It creates line by line diffs which are suited towards text files.

Bears can handle binary files by setting `USE_RAW_FILES` to True
However these bears will have to be in charge of managing the file (opening the file, closing the file, reading the file, etc). There is not a binary diff handling system set in place whose functionality the bears could use to handle binary files automatically.

Extending support for binary files will open new doors for coala to fix and lint a new wide variety of file extensions which are written in binary code (Image Files, Audio Files, Zip Files) efficiently.

Having a proper binary diff handling system along with our unified diff handling system can help minimize the codebase and also increase the

efficiency of such bears which handle binary files. Which were previously hard to write.

XML Diff

XML is a software and hardware independent markup language that defines a set of rules for encoding documents in a format that is both human-readable and machine-readable. Due to its simplicity in data sharing, transport and data availability across the Internet, it is widely used for the representation of arbitrary data structures such as those used in web services.

My project would add support for a new type of diff handling aimed specifically for XML. This diff system can be later tweaked for handling some other popular markup languages (CML, MathML, YAML etc.)

Project Milestones

My Google Summer of Code Project mainly aims at achieving the following things:

- Adding Support for XML and Binary Diffs and writing at least one bear for each
- Improving existing Diff Handling (Adding Context)
- Improving FileFactory: (Writing tests for handling of files in utf16/32 encoding, , EOL Support)

Community Bonding:

During the community bonding period, I would write my first blog post related to my GSoC Journey. I would also merge a cEP for my project. I would like to discuss with my mentor the various issues I might face forehand and also finalising a detailed project plan and timeline, thus kickstarting my GSOC journey.

I would also plan the milestones for phase 1 and become familiar with the codebase which I would be dealing with. Mainly my work would be concerned with Diff Handling, so I will become familiar with the existing diff handling system and also devise a plan on how to implement support for binary and xml diffs.

By the end of the bonding phase I would

- **Come up with a new design for xml diffs and at least one bear which uses it**
- **Come up with a design for handling binary formats and at least one bear which makes use of it**
- **Merge my cEP**
- **Come up with ideas to improve the output of our current diff handling system by showing context for the code (Parent Trees)**
- **Plan the milestones for Phase 1**
- **Plan the fixes to improve FileFactory codebase (non Unix EOL support, Handling utf16/32 encoded files testing etc.)**

Phase 1:

1. Look at relevant XML and Binary Diff Libraries. Check the following to decide the library to be chosen

    - Last Merged PRs
    - How many people involved
    - Library Well Maintained
    - Compatibility Issues (Python 3.4 Support)
    - UI not the only thing important, also the output of text file matters.

    I would start by sending a small patch to each of the relevant libraries I found. This is done to choose liabilities and check that the library is maintained up to date.

    Comparing the following data we can come to a decision on what is the right diff library to be chosen. This is done to avoid wasting time fixing bugs which would arise due to fault in their code. .

    The following are a list of some libraries which can be
    Considered:

XML Diff Libraries -

1. https://github.com/Shoobx/xmldiff
2. https://github.com/christian-oudard/htmltreediff
3. https://github.com/JoshData/xml_diff

Binary Diff Libraries -

1. https://github.com/juhakivekas/multidiff
2. https://github.com/joxeankoret/pigaios
3. https://github.com/nirizr/rematch
4. https://github.com/debasishm89/MassDiffer
5. https://github.com/CapeLeidokos/elf_diff

FileFactory:

The file proxy is a wrapper object that contains properties about the file currently processed. It contains -

- The filename, this is the starting point for every other attribute.
- The content of the file as a string.
- The content of the file, line-splitted.

FileFactory is responsible for providing access to contents of files to the bears and bear-writers and also provides methods to update the in memory content register of the said file.

I would improve FileFactory by fixing the following things

- Write tests for handling files in utf16/32 encoding
- Non Unix EOL Support

coala only handles utf-8 text encoding but windows typically uses utf16/32.

It would be good to have some tests created for handling of files in utf16/32 encoding, especially handling of patch files in utf16/32

Python has os.linesep as an OS dependent constant, and when opening a file in text-mode, it's possible to specify the expected newline sequence or by default use 'universal newlines' which will accept any of '\n' (unixy), '\r' (old-school MacOS) and '\r\n' (DOS/windows)

We have currently only specified Unix based newline sequences. To add support for non unix newlines, tests for files with EOL markers ‘/r’ and ‘/r/n’ would also need to be added.

Context for Diffs:

Currently the diffs only show the line of the code for which the diff is shown with no context as to which function the code belongs to:

```
src\add.py
[  13] ··print(data)
**** PEP8Bear [Section: cli | Severity: NORMAL] ****
!    ! The code does not comply to PEP8.
!    ! +1 -1 in c:\users\username\add.py
[    ] *0. Do (N)othing
[    ]  1. (O)pen file
[    ]  2. (A)pply patch
[    ]  3. (S)how patch
[    ]  4. Add (I)gnore comment
[    ]  5. Show Applied (P)atches
[    ]  6. (G)enerate patches
[    ] Enter number (Ctrl-Z to exit):
```

Showing some context would help the users in finding the code easily.

Git gives the function definition using `git show` which looks like this

```
@@ -77,7 +77,7 @@ def run_coala(log_printer=None,

         config_file = os.path.abspath(str(sections["default"].get("config")))

-        settings_hash = get_settings_hash(sections)
+        settings_hash = get_settings_hash(sections, targets)
         flush_cache = bool(sections["default"].get("flush_cache", False) or
                            settings_changed(log_printer, settings_hash))
```

It is way easier to understand what the code does because git gives information regarding what function it resides in even though the function definition would normally not be in the diff.

We want a similar behavior in our coala output

There are 2 types of context which can be added - extra lines before the line, and info about parents in the syntax tree (i.e class and function name).

I will add context in a sequential order - Firstly showing extra lines before and after the line, and then removing redundant info and just showing the parents.

For the first type of context, I would determine the context by checking the indentation and print all lines that have a lower indentation than the last printed one

The following approach would be used:

- Get the line number and filename of the diff from sourcerange.
- Read the file and store it as a string in a variable
- Get to the line number of diff.
- Then loop backward and compare the indentation of nth line with (n-1)th line, if equal, do nothing, if lesser, then print that line with its line number. Break out of the loop when there is a line that matches r'\n\S.*'

The function `get_context` to get the context would be written in ConsoleInteraction.py is given below

```
def get_context(file_dict,
                sourcerange):
    """
    Processes the affected line and the file to find the context for those
    lines.
    :param file_dict:             A dictionary containing all files as values with
                                  filenames as key.
    :param sourcerange:           The SourceRange object referring to the related
                                  lines to print.
    :return:                      Returns affected_line's contextual information
    """
    related_lines = list()
    related_line_nr = list()

    affected_line = file_dict[sourcerange.file][
        sourcerange.start.line - 1].rstrip()
    affected_line = affected_line.replace('\t', 8*' ')
    indent_space_count = len(affected_line) - len(affected_line.lstrip())

    if indent_space_count > 0:
        for j in range(sourcerange.start.line-2, -1, -1):    # pragma: no branch
            rline = file_dict[sourcerange.file][j].rstrip()
            affected_line = affected_line.replace('\t', 8*' ')
            prv_space_count = len(rline) - len(rline.lstrip())
            if prv_space_count < indent_space_count and rline:
                related_lines.append(rline)
                related_line_nr.append(j+1)
                indent_space_count = prv_space_count
            if indent_space_count == 0:
                break
    else:
        pre_context = list(range(sourcerange.start.line-1))
        pre_context.reverse()
        pre_context = pre_context[0: min(2, len(pre_context))]
        for i in pre_context:
            related_lines.append(file_dict[sourcerange.file][i])
            related_line_nr.append(i+1)

    return related_lines, related_line_nr
```

The output will print the lines before the code. It will look like this:

```
src\add.py
[  10] »class·helloworld(self):
[  11] »def·add(a,·b):
[  12] »data·=·'hello'
[  13] ··print(data)
**** PEP8Bear [Section: cli | Severity: NORMAL] ****
!    ! The code does not comply to PEP8.
!    ! +1 -1 in c:\users\add.py
[    ] *0. Do (N)othing
[    ]  1. (O)pen file
[    ]  2. (A)pply patch
[    ]  3. (S)how patch
[    ]  4. Add (I)gnore comment
[    ]  5. Show Applied (P)atches
[    ]  6. (G)enerate patches
[    ] Enter number (Ctrl-Z to exit):
```

For getting the Parents as context I will research some python libraries which give a tree of context based on indentation levels.

The library can then be trained to work without needing to parse the language, by including a list of keywords which indicate a significant nesting when they appear in a line (e.g. class/function for python, but also similar keywords in lots of other languages would need to be added.

With the new context the diffs will look like this:

```
src\add.py
[  10] »class·helloworld(self):
[  11] »def·add(a,·b):
[  13] ··print(data)
**** PEP8Bear [Section: cli | Severity: NORMAL] ****
!    ! The code does not comply to PEP8.
!    ! +1 -1 in c:\users\username\add.py
[    ] *0. Do (N)othing
[    ]  1. (O)pen file
[    ]  2. (A)pply patch
[    ]  3. (S)how patch
[    ]  4. Add (I)gnore comment
[    ]  5. Show Applied (P)atches
[    ]  6. (G)enerate patches
[    ] Enter number (Ctrl-Z to exit):
```

During the First Phase I plan to complete the following

- **Decide the library which would be used for generating XML and Binary Diffs**
- **Give Context in Diffs (Parents in Syntax Tree)**
- **Improve FileFactory to handle files with utf16/32 and latin1 text encodings and**
- **Write Weekly Reports mentioning my progress for each week**
- **Plan the milestones for Phase 2**

Phase 2:

The main focus of this phase will be designing of binary diffs.

A binary file is defined by the absence of an end-of-line marker.

For a binary diff, there are no lines, hence we can’t use our existing unified diff system for diffing binary files.

To show a binary diff , the UI needs to show byte-ranges of changes, using some textual encoding of the bytes (e.g. hex) with some extra bytes each side to help the user see where those bytes are . This can then be made more user-friendly by describing the change to the user with text and then do the binary change if accepted.

To do a binary diff, classes like SourceRange , etc are all problematic.
they all assume there is a end-of-line separator

I will be testing the new binary diff system on binary formats like JPEG and PNG Files.

The following process would be followed to generate binary diffs:

1. Open the original file and the fixed file.
2. Iterate through both files from the start to the end - this gives us an address (offset) in the file (byte 0, byte 1, byte 2, etc)
3. compare the bytes at the current position, and if they are not the same, print the address and the two different bytes

Currently coala has the option to run raw files. Processing allows raw files if the Bear states that it uses raw files. Bears that use binary files can't be mixed with Bears that use text files. If this is enabled, bears are incharge of doing the file handling (opening it, closing it, reading it, etc).

Bears which handle binary files are required to set USE_RAW_FILES=TRUE - coalib has code for this, but we have no bears which use this yet

Currently There are already some PRs for binary bears -

- ImageCompressionBear
- ImageDimensionBear
- MP3CheckBear

Bears do automatic improvements to files. so we need to think of an improvement to a binary file that could be automated

Binary Diffs

Let us take 2 example examples of binary files - bin_file1 and bin_file2

Below is the hexdump of the 2 binary files

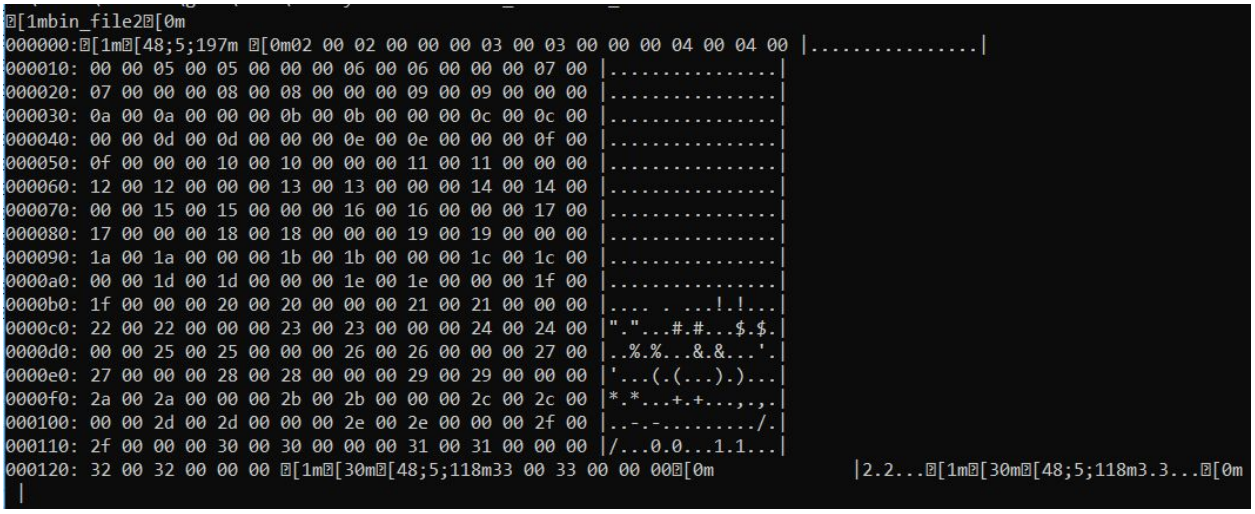
bin_file1

```
000000  01 00 01 00 00 00 02 00 02 00 00 00 03 00 03 00  ................
000010  00 00 04 00 04 00 00 00 05 00 05 00 00 00 06 00  ................
000020  06 00 00 00 07 00 07 00 00 00 08 00 08 00 00 00  ................
000030  09 00 09 00 00 00 0a 00 0a 00 00 00 0b 00 0b 00  ................
000040  00 00 0c 00 0c 00 00 00 0d 00 0d 00 00 00 0e 00  ................
000050  0e 00 00 00 0f 00 0f 00 00 00 10 00 10 00 00 00  ................
000060  11 00 11 00 00 00 12 00 12 00 00 00 13 00 13 00  ................
000070  00 00 14 00 14 00 00 00 15 00 15 00 00 00 16 00  ................
000080  16 00 00 00 17 00 17 00 00 00 18 00 18 00 00 00  ................
000090  19 00 19 00 00 00 1a 00 1a 00 00 00 1b 00 1b 00  ................
0000a0  00 00 1c 00 1c 00 00 00 1d 00 1d 00 00 00 1e 00  ................
0000b0  1e 00 00 00 1f 00 1f 00 00 00 20 00 20 00 00 00  .......... . ...
0000c0  21 00 21 00 00 00 22 00 22 00 00 00 23 00 23 00  !.!..."."...#.#.
0000d0  00 00 24 00 24 00 00 00 25 00 25 00 00 00 26 00  ..$.$...%.%...&.
0000e0  26 00 00 00 27 00 27 00 00 00 28 00 28 00 00 00  &...'.'...(.(...
0000f0  29 00 29 00 00 00 2a 00 2a 00 00 00 2b 00 2b 00  ).)...*.*...+.+.
000100  00 00 2c 00 2c 00 00 00 2d 00 2d 00 00 00 2e 00  ..,.,...-.-.....
000110  2e 00 00 00 2f 00 2f 00 00 00 30 00 30 00 00 00  ...././...0.0...
000120  31 00 31 00 00 00 32 00 32 00 00 00              1.1...2.2...
```

bin_file2

```
000000  02 00 02 00 00 00 03 00 03 00 00 00 04 00 04 00  ................
000010  00 00 05 00 05 00 00 00 06 00 06 00 00 00 07 00  ................
000020  07 00 00 00 08 00 08 00 00 00 09 00 09 00 00 00  ................
000030  0a 00 0a 00 00 00 0b 00 0b 00 00 00 0c 00 0c 00  ................
000040  00 00 0d 00 0d 00 00 00 0e 00 0e 00 00 00 0f 00  ................
000050  0f 00 00 00 10 00 10 00 00 00 11 00 11 00 00 00  ................
000060  12 00 12 00 00 00 13 00 13 00 00 00 14 00 14 00  ................
000070  00 00 15 00 15 00 00 00 16 00 16 00 00 00 17 00  ................
000080  17 00 00 00 18 00 18 00 00 00 19 00 19 00 00 00  ................
000090  1a 00 1a 00 00 00 1b 00 1b 00 00 00 1c 00 1c 00  ................
0000a0  00 00 1d 00 1d 00 00 00 1e 00 1e 00 00 00 1f 00  ................
0000b0  1f 00 00 00 20 00 20 00 00 00 21 00 21 00 00 00  .... . ...!.!...
0000c0  22 00 22 00 00 00 23 00 23 00 00 00 24 00 24 00  "."...#.#...$.$.
0000d0  00 00 25 00 25 00 00 00 26 00 26 00 00 00 27 00  ..%.%...&.&...'.
0000e0  27 00 00 00 28 00 28 00 00 00 29 00 29 00 00 00  '...(.(...).)...
0000f0  2a 00 2a 00 00 00 2b 00 2b 00 00 00 2c 00 2c 00  *.*...+.+...,.,.
000100  00 00 2d 00 2d 00 00 00 2e 00 2e 00 00 00 2f 00  ..-.-........./.
000110  2f 00 00 00 30 00 30 00 00 00 31 00 31 00 00 00  /...0.0...1.1...
000120  32 00 32 00 00 00 33 00 33 00 00 00              2.2...3.3...
```

On running these 2 files with a binary diffing library (multidiff) we get the following output -



This library can be trained to get the following output -

```
000000:<delete>01 00 01 00 00 00<delete>02 00 02 00 00 00 03 00 03 00 00 00 04 00 04 00
000120: 32 00 32 00 00 00 <insert>33 00 33 00 00 00<insert>
```

The binary diff shows:

- The address/offset where the byte changes have occured
- The bytes preceding and succeeding the changed byte range (context)
- The operation performed - (**<delete>, <insert>, <replace>**)

The ascii row (3rd column in the original diff output) was removed for better looking diffs but can be added again if we need a different design.

There are also some ideas for binary diff designs which could be implemented in the future for binary diffs:

```
Executing section cli...

src/bin_file
|   000000| 01 00 01 00 00 00 02 00 02 00 00 00 03 00 03 00
|         | [NORMAL] BinaryDiffBear:
|         | Offset contains following inconsistencies:
|         |
|---------|         | /path/src/bin_file
|         |     ++++| /path/src/bin_file
|   000000|         |-01 00 01 00 00 00 02 00 02 00 00 00 03 00 03 00
|         |   000000|+02 00 02 00 00 00 03 00 03 00 00 00 04 00 04 00
|   000010|   000010| 00 00 04 00 04 00 00 00 05 00 05 00 00 00 06 00
|   000020|   000020| 06 00 00 00 07 00 07 00 00 00 08 00 08 00 00 00
|   000030|   000030| 09 00 09 00 00 00 0a 00 0a 00 00 00 0b 00 0b 00

|    | *0: Do nothing
|    |  1: Open file(s)
|    |  2: Apply patch
|    |  3: Print more info
|    |  4: Add ignore comment
|    | Enter number (Ctrl-D to exit): 2
```

ImageCompressionBear is a binary bear which uses the optimage library to check if an image can be compressed and calculate the number of bytes which would be reduced if it is compressed.

A sample mockup for the bear is given here - ImageCompressionBear

For using the bear with the new binary diff handling system a possible approach is listed below:

- Run the bear with the image file which you want compressed
- The bear will check if the image can be compressed, If it can, the bear will generate a binary diff with the changes to the byte range.
- The user can then either choose to apply the patch or do nothing.

During the Second Phase I plan to complete the following

- **Design and Add Support for Binary Diffs to the codebase**
- **Write at least one bear which uses the new BInary Diff Handling system**
- **Write Fortnightly Reports mentioning my progress**
- **Plan the milestones for Phase 3**

Phase 3:

The main focus in Phase 3 would be adding support for XML Diffs and writing at least one bear which makes use of the new XML Diff Handling System.

Xml documents are widely used as containers for exchange and storage of arbitrary data in today's systems.

In order to send changes to an XML document, an entire copy of the new version must be sent, unless there is a means of indicating only the portions that have changed (patches).

This approach describes an XML patch framework that utilizes XMLPath Language (XPath) selectors. An XPath selector is used to pinpoint the specific portion of the XML that is the target for the change.

These selector values and updated new data content constitute the basis of patch operations described in this document

In addition to them, with basic **<add>, <replace>, and <remove>** directives a set of patches can be applied to update an existing target XML document. With these patch operations, a simple semantics for data oriented XML documents is achieved, that is, modifications like additions, removals, or substitutions of elements and attributes can easily be performed.

Basic Features and Requirements

In this framework, "XPath selector values" and new data content are embedded within XML elements, the names of which specify the modification to be performed: <add>, <replace>, or <remove>.

XPath selectors pinpoint the target for a change and they are expressed as attributes of these elements. The child node(s) of patch operation elements contain the new data content.

The following XPath data model node types can be added, replaced, or removed with this framework: elements, attributes, namespaces, comments, texts, and processing instructions.

Patch Operations

An XML diff document contains a collection of patch operation elements, including one or more <add>, <replace>, and <remove> elements.

These patch operations will be applied sequentially in the document order. After the first patch has been applied to update a target XML document, the patched XML document becomes a new independent XML document

against which the next patch will be applied. This procedure repeats until all patches have successfully been processed.

Locating the target of the patch

Each patch operation element contains a 'sel' attribute. The value of this attribute is an XPath selector. The 'sel' value is used to locate a single unique target node from the target XML document.

This located node pinpoints the target for a change and usually it is an element, which is for example either updated itself or some child node(s) are added into it. It MAY also be, for instance, a comment node, after which some other sibling node(s) are inserted.

In any case, it is an error condition if multiple nodes are found during the evaluation of this selector value.

<add> Element

The <add> element represents the addition of some new content to the target XML document: for example, a new element can be appended into an existing element.

The new data content exists as the child node(s) of the <add> element. When adding attributes and namespaces, the child node of the <add> element MUST be a single text node. Otherwise, the <add> element MAY contain any mixture of element, text, comment or processing instruction nodes in any order. All children of the <add> element are then copied into a target XML document.

The <add> element type has three attributes: **sel**, **type**, and **pos**.

The **type** attribute is an XPath selector, but it only locates attributes and namespaces.

The value of the optional **pos** attribute indicates the positioning of new data content. It is not used when adding attributes or namespaces.

When neither **type** nor **pos** attribute exist, the children of the <add> element are then appended as the last child node(s) of the located target element.

When the value of **pos** attribute is "prepend" the new node(s) are added as the first child node(s) of the located target element. With the value of "before", the added new node(s) MUST be the immediate preceding sibling node(s), and with "after", the immediate following sibling node(s) of the located target node.

<replace> Element

The <replace> element represents a replacement operation: for example, an existing element is updated with a new element or an attribute value is replaced with a new value. This <replace> operation always updates a single node or node content at a time.

The <replace> element type only has a **sel** attribute. If the located target node is an element, a comment or a processing instruction, then the child of the <replace> element MUST also be of the same type. Otherwise, the

<replace> element MUST have text content or it MAY be empty when replacing an attribute value or a text node content.

<remove> Element

The <remove> element represents a removal operation of, for example, an existing element or an attribute.

The <remove> element type has two attributes: **sel** and **ws**. The value of the optional **ws** attribute is used to remove the possible white space text nodes that exist either as immediate following or preceding sibling nodes of the located target node. The usage of **ws** attribute is only allowed when removing other types than text, attribute and namespace nodes. If the value of **ws** is "before", the purpose is to remove the immediate preceding sibling node that MUST be a white space text node and if the value is "after", the corresponding following node. If the **ws** value is "both", both the preceding and following whitespace text nodes MUST be removed.

A sample mockup for the patching process is given below:

```
An example target XML document:

   <?xml version="1.0" encoding="UTF-8"?>
   <doc>
     <note>This is a sample document</note>
   </doc>

   An XML diff document:

   <?xml version="1.0" encoding="UTF-8"?>
   <diff>
     <add sel="doc"><foo id="ert4773">This is a new child</foo></add>
   </diff>

   A result XML document:

   <?xml version="1.0" encoding="UTF-8"?>
   <doc>
     <note>This is a sample document</note>
   <foo id="ert4773">This is a new child</foo></doc>
```

For generating the XML Diffs 2 methods will have to be written in the Diff Class:

- **xml_diff()** method to generate an xml_diff corresponding to this patch
- A Classmethod **from_xml_diff** which would create a Diff Object

After adding support for the new xmldiff I would write a new bear which would make use of the new XML Diffs

We could also add the new XML Diff Handling to XMLBear which currently uses unified Diff Handling to generate diffs.

Finally I would prepare the final documentation, Test the code to fix any minor bugs, Submit the final mentor evaluation.

During the Third Phase I plan to complete the following

- **Design and Add Support for XML Diffs to the codebase**
- **Write at least one bear which uses the new XML Diff Handling system**
- **Add no Unix EOL Support to FileFactory**
- **Write Fortnightly Reports mentioning my progress**
- **Write the Final Documentation**
- **Fix any minor bugs left**
- **Submit the Final Mentor Evaluation**

References:

1. Show Context for Diffs and Lines
2. FileFactory: Support non-Unix EOL
3. Next Gen Bear Design (FileProxy)
4. Add setting charset
5. XML Diff
6. ImageCompressionBear
7. ImageDimensionBear
8. MP3CheckBear