OSM-CAT: A software application to generate contribution summaries from OpenStreetMap XML

Peter Smith B.Sc.

Research Thesis 2012

M.Sc. in Computer Science (Software Engineering)

Department of Computer Science,

National University of Ireland,

Maynooth, County Kildare, Ireland

A thesis submitted in partial fulfilment of the requirements for the

M.Sc. in Computer Science (Software Engineering)

Supervisor: Dr. Peter Mooney

January 2012

Declaration

I hereby certify that this material, which I now submit for assessment on the program of study leading to the award of Master of Science in Software Engineering, is entirely my own work and has not been taken from the work of others save and to the extent that such work has been cited and acknowledged within the text of my work.

Signed:__________________________ Date:__________________________

Acknowledgement

I would like to thank Dr. Peter Mooney at National University of Ireland, Maynooth (NUIM) for his support, guidance, and input to the project. I would also like to extend a general thank you to all of my lecturers at NUIM, for the knowledge they so generously imparted to me.

Abstract

OpenStreetMap (OSM) is currently the most extensive and widely-used example of Volunteered Geographical Information (VGI) available on the Internet. The aim of the OSM project is to provide a free and openly accessible spatial database. The data is provided by volunteers, who collect and contribute it to the OSM database using a variety of techniques and methods. OSM data is then most commonly used and accessed via a user-friendly web-based map on www.openstreetmap.org. The spatial data corresponding to any OSM mapped area can be exported in a special XML based format, namely OSM-XML. This provides a convenient and dedicated transport format which matches the OSM databases' model. Using these OSM-XML files one should be able to extract information about contribution patterns and tagging summaries for the data. However, the simplicity of OSM-XML is also potentially its greatest disadvantage. Processing OSM-XML data files efficiently can be problematic given that mapped areas can produce complex, large files.

In this thesis we present the design and implementation of a new Java-based software application called the ***O**pen**S**treet**M**ap **C**ontributor **A**nalysis **T**ool* (**OSM-CAT**) for computing contribution summaries from OSM-XML. OSM-CAT allows users to process OSM-XML data efficiently, and automatically produces a detailed summary of the contents of the dataset. This analysis places specific emphasis on 'interesting' statistics, such as *who* contributed to the OSM data in a chosen area, *what* types of contributions were made to the OSM data, *when* these contributions were made, and the accuracy of map feature tagging. While similar tools exist that do some of these tasks, OSM-CAT provides GIS researchers and interested individuals with a complete and integrated overview of contributions to OSM, corresponding to the input OSM-XML datasets. We present a full analysis of OSM-CAT on a large set of OSM-XML datasets, and discuss its usefulness to the OSM community and beyond. We close the thesis with some conclusions, and set out a number of issues for consideration as future work. A comprehensive appendix is provided with additional information for those wishing to use OSM-CAT.

Table of contents

1. Introduction 1

2. Background & Proposal 3

  2.1 The OpenStreetMap XML Format (OSM-XML) 3

  2.2 Problem Analysis 5

  2.3 Related Work 9

  2.4 Our Proposal 13

  2.5 Software Libraries & Frameworks Used 14

3. Software Design 17

  3.1 OSM-CAT Design Overview 17

  3.2 OSM-CAT Functionality Summary 19

  3.3 OSM-XML Selective Parser 19

  3.4 Graphical User Interface (GUI) 24

  3.5 Report Output Formats 25

4. Implementation 27

  4.1 OSM-CAT Software Modules Overview 27

  4.2 Input and Output (I/O) 28

  4.3 Data Storage/Data Structures 32

  4.4 Graphical User Interface (GUI) 36

  4.5 Utilities 41

5. Experimental Analysis 44

  5.1 Experimental Setup 44

  5.2 Results 47

6. Conclusion 55

  6.1 Critical Analysis 55

  6.2 Future Work 60

Bibliography 63

Appendix A: UML Class Diagrams 66

  A.1: OSM-CAT UML Class Diagram 66

  A.2: I/O Module UML Class Diagram 67

  A.3: Data and MapElements Modules UML Class Diagram 68

  A.4: GUI Module UML Class Diagram 69

  A.5: Utilities Module UML Class Diagram 70

Appendix B: Java Code ... 71
B.1: iText PDF Chapter Creation ... 71
B.2: HTML Generator Section Creation ... 71
B.3: JFreeChart Chart Creation ... 72
B.4: Comparator Implementation and Use ... 72
Appendix C: Figures ... 73
C.1: Sub-section of Default XML Valid Tags File ... 73
Appendix D: OSM-CAT Graphical User Interface Overview ... 74
Appendix E: Eclipse Development Setup Guide ... 78

1. Introduction

The phenomenon of Volunteered Geographic Information (VGI) [1] is the result of a number of developments. In particular, the availability of affordable, compact, Global Positioning Systems (GPS), and the growth of rich web applications, known collectively as Web 2.0, have resulted in this new kind of geographic spatial information. Where previously the collection and processing of spatial data were tasks performed strictly by professionals, these developments gave rise to a new community of 'amateur' spatial data producers, who could independently, and often quite accurately, perform the same tasks. These Citizen Scientists[1] have introduced a new kind of sensor network and with it the idea of humans as sensors [1].

The result of this democratization of Geographic Information Systems (GIS) is a whole range of projects which use this crowdsourcing [2] of geographic spatial data in clever and innovative ways. The OpenStreetMap (OSM) [3] is probably the most extensive, and most widely used example of such a project. OSM allows people to use, and contribute to, spatial data stored in its databases. Real world geographical data is uploaded into the OSM and stored as *Nodes* (points), *Ways* (polylines[2]), and *Relations*. These geometric primitives can then be richly annotated with *tags*, allowing complex relationships to be expressed. The collection of spatial data for any specified mapped area can be exported and downloaded in an OSM specific, XML[3] [4] format, namely OSM-XML [5]. Processing and analysing this data can produce useful and important information for both researchers and anyone with an interest in spatial data. Current relevant research papers [6] [7] [8] demonstrate this usefulness, where other forms of VGI are used to great effect in various applications. Of particular interest is information relating to the contributors of OSM's spatial data in a chosen area, such as:

- *Who* contributed to the OSM data in the chosen area
- *When* these contributions were made
- *What* contributions were made
- *How much* contributed information is valid/acceptable
- *How* were the contributions made

1 A popular term for communities of citizens who act as observers in some domain of science.

2 A polyline, or polygonal chain, is a connected series of line segments.

3 The Extensible Markup Language (XML) is discussed in section 2.1

However, processing OSM-XML files of arbitrary size to obtain contributor related information can be problematic using conventional XML processors. For example, the OSM-XML file size for even a relatively small geographic region such as Paris,France[4][9] is already over 70 MB, and growing with each new contribution. Such large file sizes usually result in lengthy processing times, which may be off-putting to potential analysts. Also, a file of this size might contain a vast amount of complex spatial data. Developing an application to analyse such data would most likely require substantial effort. In our research of OSM-XML data analysis, we found that, currently, there is no application which provides a comprehensive overview of the quality of OSM-XML data, and the contributions/contributors. This is surprising, considering the current interest in OSM data among GIS researchers [1] [8] [10] [7] [11], and the widespread use of OpenStreetMap [12] by the general public.

To perform detailed analysis, we will design and implement the ***O**pen**S**treet**M**ap **C**ontributor **A**nalysis **T**ool* (**OSM-CAT**), a lightweight, efficient, software application, to efficiently process arbitrarily sized OSM-XML files. OSM-CAT will place particular emphasis on smaller regions of an OSM map – towns, city suburbs, electoral districts - producing useful contributor related data, patterns and summaries. OSM-CAT will allow researches, and indeed general OSM users, to view detailed contributor information for any mapped area. This project will produce an application which not only efficiently processes OSM-XML data, producing correct, valid analysis data, but also one which can be easily used and understood by GIS researchers, OSM enthusiasts, and the general public. It will abstract the details of OSM-XML from users, allowing them to produce detailed analysis of OSM data, without the need for an in-depth understanding of OSM-XML's structure.

The rest of this paper is organized as follows. In Section 2, we provide background into the problems of processing arbitrarily sized, complex OSM-XML files, and propose our solution to these problems. We also examine related work and look at existing software artefacts which can be used in our solution. Section 3 details the design of our solution, and our software's implementation is detailed in Section 4. Section 5 describes the experimental analysis of our work. Finally, Section 6 provides a critical analysis, presents our conclusions, and makes suggestions for future work.

4 A n OSM-XML file representing the 3 Km squared centre of Paris was used.

2. Background & Proposal

In this section we provide all the background information required to understand why we propose a software application to compute contribution summaries from OSM-XML extracts. We first present the OSM-XML file format. We describe conventional approaches that exist to process arbitrarily sized XML files, and analyse the problems which exist with such approaches. We also illustrate related work, both literature-based related work, and related software already available that can be used to process and analyse OSM spatial data. Then, we present our proposal, and describe why we propose our contributor analysis application. Finally, a brief description of the technologies and tools used in the OSM-CAT project will be provided.

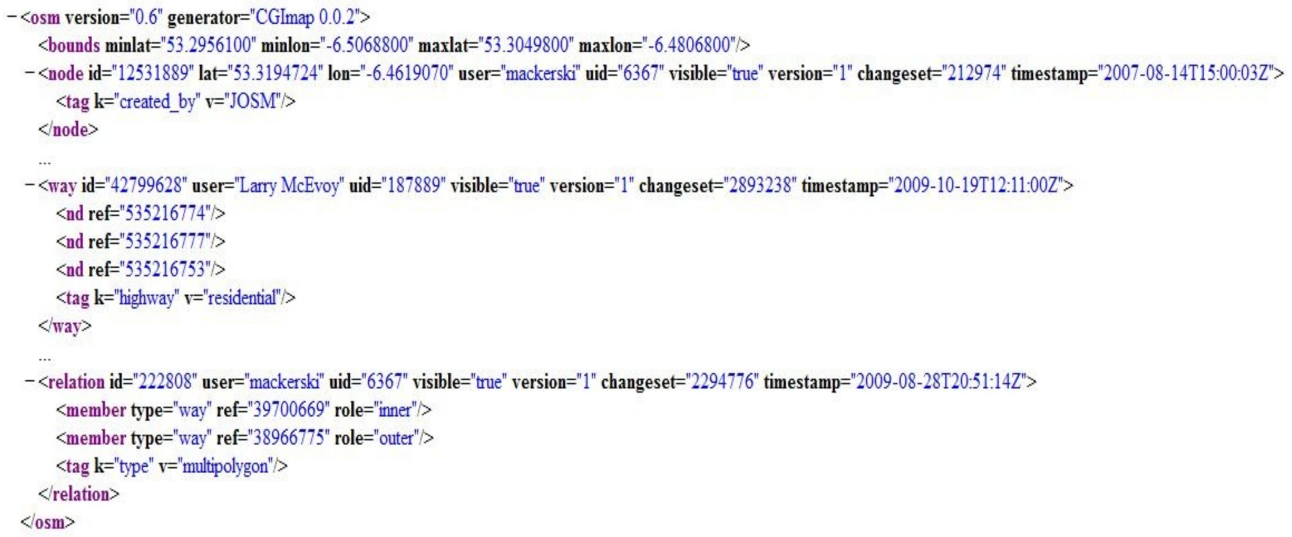
2.1 The OpenStreetMap XML Format (OSM-XML)

The Extensible Markup Language (XML) is a WC3 endorsed standard for document mark-up. This standard defines a set of rules for encoding documents in a machine-readable form, and provides a format for documents that is flexible enough to be used across a variety of domains, such as web-sites, vector graphics, and in the case of OSM, spatial data. Essentially, XML uses structured tags to form a hierarchy of elements. It is a particularly useful format, as the data represented in XML files can be processed and manipulated by a range of programming languages such as Java, C, Python and Perl. Indeed, Many Application Programming Interfaces (APIs) have been developed that process XML data, and several XML schema systems exist to aid in the definition of XML-based languages. An XML schema describes the structure of an XML document.

Programming languages, such as those mentioned above, traditionally use an XML parser to read, process, and manipulate a markup language such as XML. Parsing, or syntactic analysis, is the process of analysing a text to determine its structure. Specifically, a parser is responsible for dividing a document into individual elements, attributes, and other parts. By doing so, a parser converts an XML document into a internal representation of the XML data. How this parsing is done, and the output of this parsing, depends on the type of parser used.

There are two basic ways in which programming languages interact with XML – the Simple API for XML (SAX) [13], and the Document Object Model (DOM) [14]. The SAX API is used to process XML by reading a file one small piece at a time. For each element that it finds, SAX calls a method. In this way, SAX models the parser. The DOM API is used to process XML by reading the entire document at once, and creating an internal representation of all the contained data. Rather than modelling the parser, DOM models the whole XML document. These two vastly different approaches to XML processing have major implications on the time required to process an arbitrarily sized XML document. In the following section, we discuss these implications in greater detail.

As previously mentioned, spatial data from any OSM mapped area can be downloaded and saved in a number of different formats. One of these formats is the OSM-XML [5] format. **Figure 2.1** presents a shortened example of a complete OSM-XML file.



```
- <osm version="0.6" generator="CGImap 0.0.2">
    <bounds minlat="53.2956100" minlon="-6.5068800" maxlat="53.3049800" maxlon="-6.4806800"/>
  - <node id="12531889" lat="53.3194724" lon="-6.4619070" user="mackerski" uid="6367" visible="true" version="1" changeset="212974" timestamp="2007-08-14T15:00:03Z">
      <tag k="created_by" v="JOSM"/>
    </node>
    ...
  - <way id="42799628" user="Larry McEvoy" uid="187889" visible="true" version="1" changeset="2893238" timestamp="2009-10-19T12:11:00Z">
      <nd ref="535216774"/>
      <nd ref="535216777"/>
      <nd ref="535216753"/>
      <tag k="highway" v="residential"/>
    </way>
    ...
  - <relation id="222808" user="mackerski" uid="6367" visible="true" version="1" changeset="2294776" timestamp="2009-08-28T20:51:14Z">
      <member type="way" ref="39700669" role="inner"/>
      <member type="way" ref="38966775" role="outer"/>
      <tag k="type" v="multipolygon"/>
    </relation>
  </osm>
```

Figure 2.1: Standard OSM-XML file format (partial example shown)

This OSM-XML format follows its own particular schema system. When the OSM database receives a request for OSM-XML data, each response is wrapped in an *<osm>* element. This wrapping *<osm>* element is returned in the form of an OSM-XML file, the structure of which is as follows:

- An *<osm>* element, containing the API version, and the generator (tool) used
    - A block of *Nodes* - Each Node has the form *<node>...</node>*
    - A block of *Ways* - Each Way has the form *<way>...</way>*
        - References to the Way's Nodes
    - A block of *Relations* - Each Relation has the form*<relation>...</relation>*
        - References to the Relation's members

This structure is shown in **Figure 2.1.** We see that an OSM-XML file contains a list of OSM data primitive instances– *Nodes*, *Ways* and *Relations* [15]. Nodes define a single *geospatial* point using a latitude and longitude value. They are used to define standalone point features, which can then be used to define the path of a Way. A Way is an ordered list of between 2 and 2,000 Nodes. Multiple Ways can be used to represent extensive features, and closed Ways (*Polygons*) represent areas. A Relation is used to group data primitives – either Nodes, Ways, or other Relations – in an element which should have a value, or Tag, associated with it. These primitives represent OSM spatial data, and graphical maps are rendered directly from this data.

Associated with each primitive element in an OSM-XML file is a *user* attribute and a *uid* attribute. These are shown in **Figure 2.1**. A *user* indicates the OSM user name of the volunteer who contributed the VGI to OSM. The *uid* represents the user's unique OSM id. As OSM is made possible entirely by volunteer contributions, users of our proposed OSM-CAT software will be particularly interested in information, patterns and statistics pertaining to these volunteer contributors.

2.2 Problem Analysis

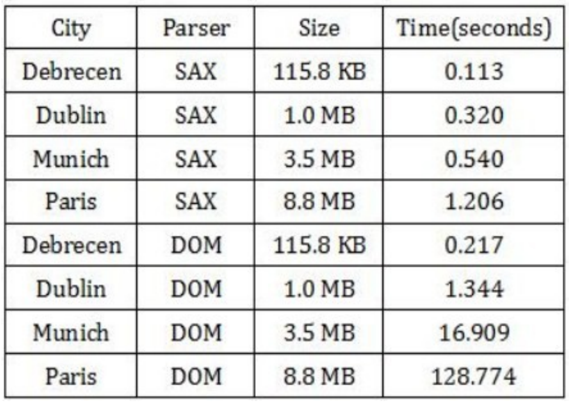
OSM is a widely used [12], collaborative project, which aims to provide a free, user-created, map of the world. OSM users volunteer spatial information, which is added to the OSM database. This data is not only used by OSM to create free 2D maps, but is freely available, and can be accessed and used by anyone. Consequently, issues such as data integrity, quality, and validity are of great importance. Such issues continue to be central topics among both OSM users and OSM researchers [16] [11] [10]. To investigate such issues, OSM data needs to be analysed. However, analysis of OSM data, and OSM-XML data in particular, is not trivial, and efficient analysis applications are required.

OSM maps generally encompass a substantial amount of information. The efficient processing of arbitrarily-sized OSM-XML files is intrinsically hard to perform, due to both the possible large sizes of the files being processed, and to the variety and complexity of the data represented in these files. The enormous wealth of OSM spatial information that can be contained within even the smallest of maps, dictates that representing even relatively small areas such as villages and towns can produce large OSM-XML files. We catalogue the problems associated with OSM-XML file processing, which we discovered through our research of OSM-XML files.

2.2.1 File Sizes & Processing Times

To investigate the problems associated with processing arbitrarily sized OSM-XML files [17] [18], we selected four diverse European cities. Using the longitude and latitude coordinates for these four cities, taken from each city's Wikipedia website, we downloaded the OSM-XML file for a 1 Km squared grid, based on the centroid[5] coordinates of each city. Specifically, we selected Debrecen, Hungary [19], with a population of 208,016, and a file size of 115.8 KB, Dublin, Ireland [20], with a population of 525,383, and a file size of 1.0 MB, Munich, Germany [21], with a population of 1,353,186, and a file size of 3.5 MB, and finally Paris, France [22], with a population of 2,211,297, and a file size of 8.8 MB.

We then performed experimental analysis on the selected OSM-XML files. We first created two simple Java OSM-XML parsers – one using SAX, the other using DOM. Using the SAX and DOM parsers we performed basic parsing tasks such as extracting and displaying all the OSM contributor names and Ids associated with each OSM primitive represented in the four OSM-XML files. The results of our analysis[6] are shown in **Figure 2.2**.



City	Parser	Size	Time(seconds)
Debrecen	SAX	115.8 KB	0.113
Dublin	SAX	1.0 MB	0.320
Munich	SAX	3.5 MB	0.540
Paris	SAX	8.8 MB	1.206
Debrecen	DOM	115.8 KB	0.217
Dublin	DOM	1.0 MB	1.344
Munich	DOM	3.5 MB	16.909
Paris	DOM	8.8 MB	128.774

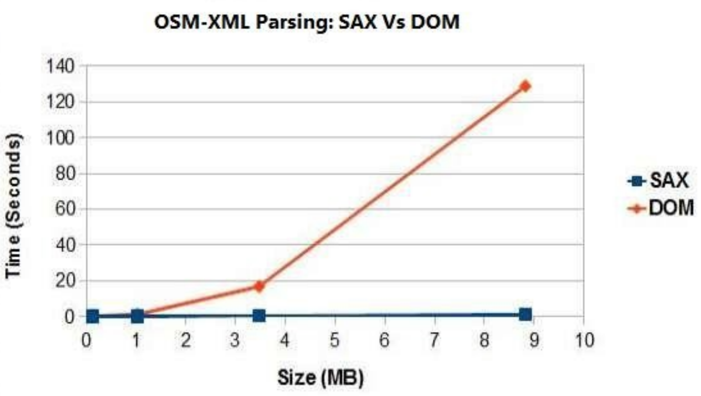


Figure 2.2: SAX & DOM OSM-XML parsing comparison

As we can see from the results, the time taken to process an OSM-XML file can vary greatly depending on the file size and the parsing API used. As shown in **Figure 2.2**, the increase in time taken for the SAX-based parser is extremely small in relation to the increase in file size,

5 In geometry, the centroid coordinates of a two-dimensional shape is the intersection of all straight lines that divide the shape into two parts of equal moment about the line.

6 All experimental analysis referenced in this paper is performed using a single desktop machine running a quad core Intel i5 2.67GHz processor with 6GB of RAM.

while for the DOM-based parser, the increase in time taken is quite substantial in relation to the increase in file size. Processing times using the DOM parser greatly increase as the file size grows. This of course would suggest that SAX is the more suitable parsing API for our proposed analysis application.

2.2.2 Complexity of OSM-XML Data

Examining the four OSM-XML files from Section 2.2.1, we find that representing even relatively small areas requires a complex mixture of data types and values. For the purposes of our study, we will define a method to express the complexity of OSM-XML files. In our opinion, an OSM-XML file's complexity is determined by its contents, or composition, and so we call this measure of complexity the *File Composition Measure (FCM).* By defining our FCM, we will have a metric which allows us to quickly assess the complexity of any OSM-XML file. Also, this metric will be used in the assessment of our data analysis application. It must be noted that our definition of what constitutes complexity is not definitive, given that the measure of an OSM-XML file's complexity may be open to subjective interpretation. However, we feel our decision to define our FCM is justified, keeping in mind that we know of no existing metrics for measuring OSM-XML file complexity, and that previous known works have not attempted to define such a metric. We define our OSM-XML File Composition Measure below.

When an OSM-XML file is requested for a specific mapped area, an XML file containing all the current OSM spatial data is returned. This data consists of the OSM primitives – Nodes (N), Ways (W), and Relations (R), which together define the mapped area. Examining the OSM database statistics [23], we find that the overall OSM database primitive count is divided as follows; N – 1,333,555,856, W – 121,595,611, R – 1,256,916. Therefore, out of a total primitive count of 1,456,408,383, the percentage of each primitive type is; N – 91.6%, W - 8.3%, R – 0.1%. We note that the percentage of Relations (R) is particularly small. As described in Section 2.1, Relations are logical groupings of OSM primitives. As such, they are generally not directly imported into databases and GIS systems [24]. With this in mind, and the fact that the percentage of Relations in the OSM database is so negligible – 0.1%, we decide not to include Relations in the definition of our FCM.

To define our FCM, a fixed *ceiling* (upper) value is used and the complexity of an OSM-XML file is expressed against it. Taking the four 1 Km square grid OSM-XML files we selected in Section

2.2.1, we find that Paris,France is the largest (8.8 MB), and therefore contains the most data. Using our simple SAX parser from Section 2.2.1, we find that the Paris file contains 36,198 Nodes (N) and 4,470 Ways (W). Therefore, we take N_c = 37,000 and W_c = 5,000 as being the ceiling value for the number of N and W that a 1 Km square grid can contain. We increase N_c and W_c slightly to account for potential additional Nodes and Ways. This is referred to as *Normalisation* [25]. Then, to measure the FCM of the Paris file, we calculate N/N_c = 0.9783 and W/W_c = 0.8940, and add them together. This gives the Paris OSM-XML file an FCM of 1.8723. Because Paris is the largest of the four selected files, none of the other files will ever get an FCM larger than 2.0. The lowest value of FCM is 0 – but this is impossible as there would be no data. Now we have our FCM value, which is constrained within the range FCM = [0,2.0].

With our method for calculating the FCM of OSM-XML files defined, we re-examine the four OSM-XML files from Section 2.2.1. These four files represent four diverse cities, in terms of both population, and location. Having already calculated the FCM of Paris, we calculate the FCM of the three remaining cities. The FCM of each city is shown in **Table 2.1**.

File	Size	File Composition Measure (FCM)
Debrecen, Hungary	115.8 KB	0.0357
Dublin, Ireland	1.0 MB	0.2672
Munich, Germany	3.5 MB	0.8931
Paris, France	8.8 MB	1.8723

Table 2.1: File Composition Measure of 1 Km square grid OSM-XML files for selected cities

In our opinion, the results show that even a small selection of OSM-XML files, representing areas of just 1 Km squared, can produce a wide range of FCMs. The FCMs shown above vary greatly, considering our FCM is constrained within the bounds [0,2.0]. Thus, we need to ensure that our data analysis application is capable of efficiently processing OSM-XML files of varying, arbitrary FCMs.

We have discussed two problems associated with OSM-XML file processing – file size and file complexity. Also, we have defined a metric for quantifying the complexity of OSM-XML files – our File Composition Measure. Our FCM is further expanded in Section 5.1, where it is used in the analysis of our work. In the following section, we examine related work.

2.3 Related Work

In this section we provide an overview of related work. We have divided related work into literature-based related work, and software-based related work. Software-based related work refers to relevant software tools, which may not have peer-reviewed literature related to them.

2.3.1 Literature-based Related Work

OSM data processing, and analysis of VGI, is an active research topic [10] [16] [7] [11] [8]. Numerous mature and experimental ideas have been proposed in recent years to use VGI in innovative and exciting ways. There is also great interest in the information that is volunteered to OSM, with questions being asked about the integrity and quality of the OSM dataset. We intend to design and implement an application which will enable anybody to process and analyse OSM VGI. Although we are more concerned with the software which will provide analysis capabilities, as opposed to the implications and usage of the information produced by the analysis, it is important to examine related work. By doing so, we demonstrate the great need for, and usefulness of, an OSM data analysis application.

Quality assessment of the OSM dataset is an important topic, as it has implications on how the data should be interpreted, and where it should be used. Girres and Touya [10] examine the quality of the French OSM dataset. They analyse the dataset using a selection of quality metrics such as geometric accuracy, attribute accuracy, and semantic accuracy. To assess spatial data they sample various areas in France, based on the types of spatial data available in each area, such as mountains, plains, roads, buildings etc. Some quality elements are also assessed with respect to the entire French OSM dataset. They conclude that their assessment "reveals the key role of specifications to ensure quality", as errors arise due to lack of, or inconsistent, specifications.

Flanagin and Metzger [16] examine VGI with regard to its "quality, reliability, and overall value". They analyse the credibility of VGI, and suggest that credibility is inherently difficult to measure with this method of information gathering. They conclude that while VGI has a profound, and possibly positive impact on geospatial knowledge, of critical importance are "issues of information and source credibility, and their relation to the production and use of VGI".

Mooney and Corcoran [7] investigate the annotating, or tagging, of spatial objects from the OSM databases. They analyse 25,000 heavily edited objects from the databases of Ireland, United Kingdom, Germany and Austria. They present the idea that exhaustive tagging is time-consuming and labour intensive, but also important for the quality of the VGI. Overall, they conclude that "there are issues in how contributors tag and annotate spatial features in OSM", and that these issues need to be addressed "before OSM can be considered for use in serious geomatics[7] applications".

Moreover, Mooney, Corcoran et al. [11] investigate OSM data to develop measures of quality for OSM which operate in an unsupervised manner. They provide results of analysis of OSM data from several cities. They find that cities and towns lend themselves to data gathering, while rural areas require greater effort. In conclusion, they state that "the GIS community will require strong evidence of the quality of OSM data", and finally, that "OSM specific metrics are required".

M.Over et al. [8] investigate the generation of 3D models based on the free spatial data available from the OSM dataset. In particular, they review the suitability and quality of available data. With respect to the quality of OSM data, they find that although anyone can upload map data to the OSM, "due to the large number of editors, errors and conflicts are usually quickly resolved". They also state that OSM has "probably the most up-to-date map data", when compared to other map providers' data. However, they conclude that the suitability of OSM data must be further investigated, "due to the pronounced regional differences in the quality of the dataset".

2.3.2 Software-based Related Work

OSM mapping projects [26] are ongoing in most parts of the world, with many of these projects involving the creation of new computer software to process, analyse, and manipulate OSM data. Of particular interest are a number of tools currently available which provide some analysis of OpenStreetMap contribution-related information. We discuss four of these tools in this section. Although most of these tools don't directly process OSM-XML data, with the exception of *Osmosis*, they are of interest due to the information produced by their analysis of OSM data. Information regarding these and other OSM statistical related tools can be found on

7 Geomatics is the discipline of gathering, storing, processing, and delivering geographic data.

the OpenStreetMap statistics page [12].

AltogetherLost [27] provides a graphical interface to statistics generated about the global OpenStreetMap database, and provides some information on OSM contributors, such as the number of members who are the last modifiers of at least one OSM primitive in the previous day. **Figure 2.3** shows a statistical chart generated by AltogetherLost.

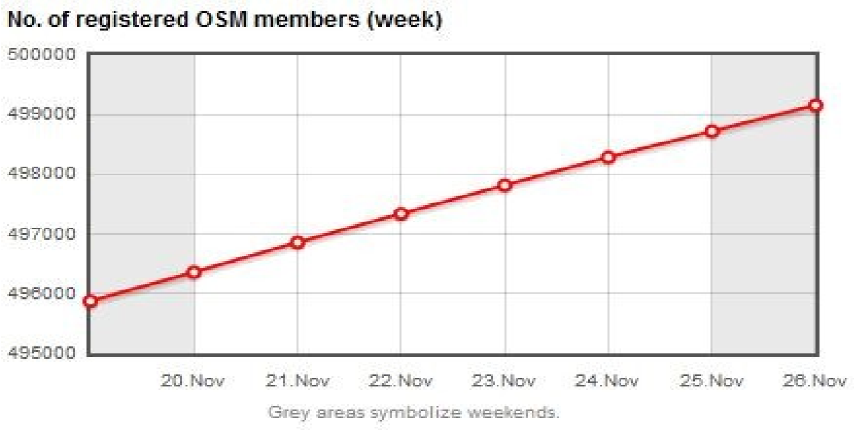
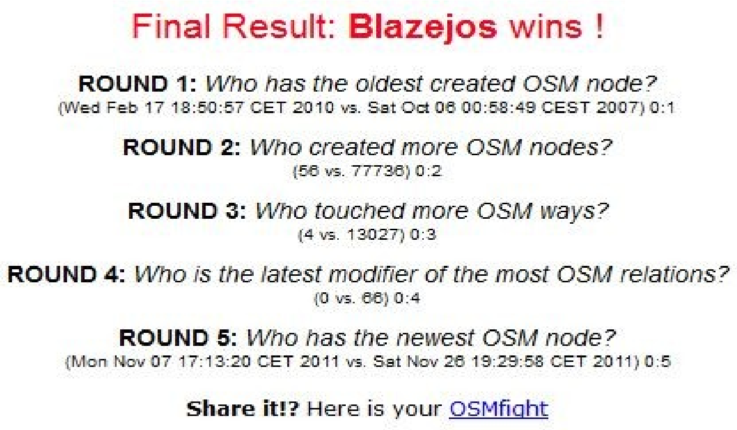


Figure 2.3: AltogetherLost generated chart

Although the statistical information generated by the tool is useful, we feel that users could greatly benefit from an application which provided statistical information of greater detail.

OSMFight [28] is an innovative, web-based tool which performs a rough quantitative comparison of two users' contributions. Two OSM users' names are entered and results are returned showing which user won each of the 'fight rounds'. **Figure 2.4** shows the result of a 'fight' between two OSM users, calculated using the entertaining OSMFight tool.



Final Result: **Blazejos** wins !

ROUND 1: *Who has the oldest created OSM node?*
(Wed Feb 17 18:50:57 CET 2010 vs. Sat Oct 06 00:58:49 CEST 2007) 0:1

ROUND 2: *Who created more OSM nodes?*
(56 vs. 77736) 0:2

ROUND 3: *Who touched more OSM ways?*
(4 vs. 13027) 0:3

ROUND 4: *Who is the latest modifier of the most OSM relations?*
(0 vs. 66) 0:4

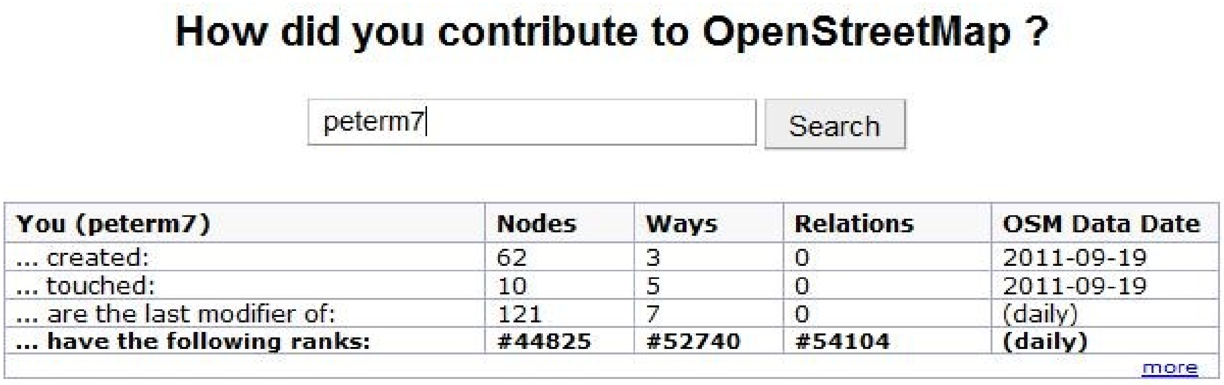
ROUND 5: *Who has the newest OSM node?*
(Mon Nov 07 17:13:20 CET 2011 vs. Sat Nov 26 19:29:58 CET 2011) 0:5

Share it!? Here is your OSMfight

Figure 2.4: OSMFight results

However, as unique and innovative as OSMFight is, in our opinion the OSM contributor related information being processed could be used to provide statistical data of greater significance.

How did you contribute to OpenStreetMap (HDYC) [29] is a web-based tool where we can find information about OpenStreetMap contributors. For example, we can see the first/last Node the contributor edited and the total amount of created Nodes, Ways, and Relations. **Figure 2.5** shows the results of a user search using HDYC.



You (peterm7)	Nodes	Ways	Relations	OSM Data Date
... created:	62	3	0	2011-09-19
... touched:	10	5	0	2011-09-19
... are the last modifier of:	121	7	0	(daily)
... have the following ranks:	**#44825**	**#52740**	**#54104**	**(daily)**

Figure 2.5: How did you Contribute results for the OSM user 'peterm7'

HDYC provides useful contributor related statistics. However, again, in our opinion, the data produced by the tool could be more detailed. Indeed, the data being processed could be utilised to provide statistics of greater use.

Osmosis [30] is a command line application for processing OSM-XML data. It is written in Java, open-source and freely available. It is of particular interest as it directly processes OSM-XML files, just as our application will do. Osmosis consists of a series of small components that can be used together to perform a larger operation. It can be used for such tasks as comparing two OSM-XML files, and extracting data from an area or a polygon. **Figure 2.6** shows the Osmosis command for dividing an area into four 'quadrants', based on latitude and longitude coordinates.

```
osmosis
  --read-xml SloveniaGarmin.osm --tee 4
  --bounding-box left=15 top=46 --write-xml SloveniaGarminSE.osm
  --bounding-box left=15 bottom=46 --write-xml SloveniaGarminNE.osm
  --bounding-box right=15 top=46 --write-xml SloveniaGarminSW.osm
  --bounding-box right=15 bottom=46 --write-xml SloveniaGarminNW.osm
```

Figure 2.6: Osmosis command line usage

Osmosis does provide useful information, but, as can be seen from **Figure 2.6**, it not only requires that the user be familiar with geographic data, but also that the user is comfortable and proficient with command line usage. Initial use of the tool requires a substantial learning curve, which may be off-putting to new users. We will seek to avoid such required knowledge and steep learning curves.

We have analysed the problems associated with processing OSM-XML files, and presented both literature-based and software-based related work. In the following section, we present our proposal for a OSM-XML data analysis application.

2.4 Our Proposal

Investigation of the OSM dataset has major advantages for both VGI and GIS, particularly:

- Crowdsourcing, which involves using a large number of people to perform tasks, is a relatively new and valuable source of VGI. Volunteers make OSM possible, and therefore the credibility of the VGI is extremely important. For OSM to be taken seriously, and indeed for the VGI to be used in other applications, it is necessary that investigations are carried out into the contributions of VGI. By analysing the contributors of the data, we can better understand who contributes, how they contribute, and how much they contribute.
- Implementation of the findings of investigations regarding the quality of the VGI submitted to OSM should lead to an increase in quality. By examining the OSM dataset, we can gain a greater understanding of what factors influence the quality of VGI. In particular, analysing what annotations contributors use when tagging, and the validity of annotations, should lead to an improvement in the tagging specifications.
- A key component of spatial data is a geographic primitive. In OSM data, three primitive types are used - Nodes, Ways and Relations. Analysing what exactly is involved in mapping a particular area, such as the quantity of each primitive, will afford a deeper understanding of not only that area, but also of the features which are most mapped by contributors, or which they consider most important. Also, investigating the validity of important spatial elements such as polygons will contribute to our understanding of quality and validity issues surrounding VGI.

However, in our opinion, the advantages stated above, which result from the analysis of OSM data, are not so easily achievable with the tools currently available. Indeed, as we have shown in Section 2.3.2, the tools currently available do not provide enough useful statistical data. To truly utilise existing OSM data, and further encourage the contribution of valuable, credible VGI, we have to ensure that powerful analysis applications are available which enable the general public to gain a deep understanding of what, when, and how they are mapping. Only by providing such a data analysis application can users take full advantage of the detailed information provided by OSM-XML files.

It is with these considerations in mind, that we propose the design and implementation of a software application to compute contribution summaries from OSM-XML. We call our application the ***O**pen**S**treet**M**ap* ***C**ontributor* ***A**nalysis* ***T**ool* (**OSM-CAT**). We propose this application with two central goals. Specifically, we will produce an application which:

1. Efficiently processes arbitrarily-sized, and arbitrarily-complex OSM-XML files, producing correct, valid analysis data.
2. Completely abstracts the details of OSM-XML data from the user, and presents them with a detailed, useful analysis of many aspects of OSM in the given mapped area.

Additionally, it must be noted that by efficient processing, we mean processing which both effectively handles any unexpected circumstances (e.g. corrupt source data, invalid source values etc.), and which requires only a reasonable amount of time. Also, as files larger than 100 MB are probably best dealt with using a database, OSM-CAT will limit analysis to files sizes no larger than 100 MB. Indeed, we will design our application to perform processing and analysis without the use of a database. In our opinion, this is a vital design goal, as a requirement for a database may discourage use of our application.

We believe that our proposed application will encourage investigation and analysis of OSM data, helping to achieve the VGI and GIS related advantages discussed at the beginning of Section 2.4. To make our proposal more concrete, in Section 2.5 we present the software libraries and frameworks that will be used in our project.

2.5 Software Libraries & Frameworks Used

In this section, we will give a brief description of the software libraries and frameworks which will be used in the implementation of our contributor analysis application.

2.5.1 JFreeChart

JFreeChart [31] is a Java chart library which allows developers to produce professional charts. It uses a flexible design, making it easy to extend, and it supports many output types, such as Java swing components, image files, and vector graphics file formats. JFreeChart is open-source and is distributed under the GNU Lesser General Public License (LGPL), which permits its use in most applications. Most importantly, its use is fully permitted in our open-source software.

2.5.2 iText

iText [32] is a Java PDF library which allows developers to create and manipulate PDF documents. Using iText, developers can perform such PDF related tasks as generating dynamic documents, and manipulating document contents. iText is an open-source software project, and is distributed under the GNU Affero General Public License (AGPL). Importantly, this means that it is suitable for use in our software application.

2.5.3 Java HTML Generator

The *Java HTML Generator* [33] is a HTML generator package for Java which allows developers to manipulate complex, nested HTML tags easily. Using this package, developers can quickly and dynamically create HTML files, for display in a web browser. It is fully open source and is distributed under the GNU General Public License (GPL), making it suitable for use in our open-source software.

2.5.4 Java OpenStreetMap Editor (JOSM)

JOSM [34] is an editor for OSM data, written in Java. It is open-source software, and its source code is freely available. Code reuse is an important aspect of software engineering, and we aim to reuse code originally developed for JOSM. This will not only be good software engineering practice, but as JOSM is a well known and respected tool, it will also encourage acceptance of our software by the OSM and GIS developer community.

2.5.5 Java Topology Suite (JTS)

The *Java Topology Suite* (JTS) [35] is a Java library of 2D spatial predicates and functions, that provides a model for linear geometry, together with a set of geometric functions. Developed by VividSolutions, its aim is to provide a complete implementation of 2D spatial algorithms. It is open-source, and is published under the GNU Lesser General Public License (LGPL). Being a well regarded Java library amongst the GIS community, its use will encourage the GIS community to use our application, and trust the underlying implementation.

In this section, we provided the background information required to understand why we propose a contributor analysis application, and we presented the proposal of our application. In Section 2.1, we discussed traditional XML processing, and in Section 2.2, we examined the various problems associated with processing arbitrarily-sized, arbitrarily-complex OSM-XML files. In Section 2.3, we presented related work, both literature-based, and software-based. In Section 2.4, we proposed OSM-CAT, our solution to the problems discussed in Section 2.3. Finally, in Section 2.5, we looked at existing software artefacts which will be used in our implementation. Next, in Section 3, we will detail the design of our data analysis application, OSM-CAT.

3. Software Design

In this section, we design OSM-CAT, so that it solves the problem of an efficient, user-friendly contributor analysis application for OSM-XML data. First, we provide an abstract overview of OSM-CAT's design, and present a summary of the functionality which OSM-CAT will provide. Then, we design the OSM-XML parser. This will be the central component of our application, and will be responsible for reading and processing the OSM-XML files. Next, our Graphical User Interface (GUI) is designed in such a way as to encourage use, and be intuitive to the general user. Finally, we present the output formats we have chosen for our analysis reports, and detail the design which allows these reports to be created.

3.1 OSM-CAT Design Overview

To convey an abstract overview of OSM-CAT's design, we present the high-level design diagram of OSM-CAT, along with its UML Use Case diagram.

3.1.1 OSM-CAT Abstract Overview

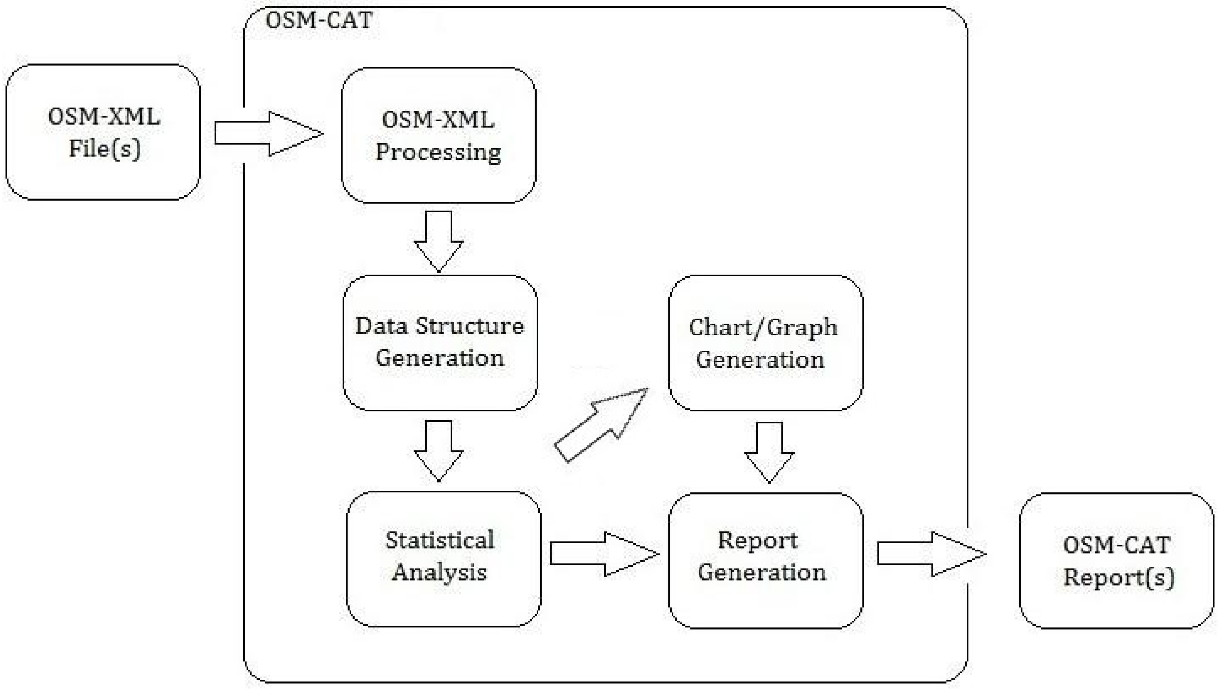


Figure 3.1: OSM-CAT abstract design overview

As we can see from **Figure 3.1**, the user will submit an OSM-XML file, or files, to OSM-CAT for analysis. The software will then process the OSM-XML file, and use the resulting data to build internal Java data structures. These data structures will roughly correspond to the OSM data contained within an OSM-XML file, including, but not limited to, Nodes, Ways, Relations and Contributors.

Once the application has prepared the necessary data structures, it will perform the following tasks, which are central to the report generation process:

1. Using the internal data structures, OSM-CAT will perform analysis, computing the required statistics and summaries for the submitted OSM-XML file.
2. OSM-CAT will then generate charts, or graphs. These charts will display the statistical data previously generated.
3. Finally, the requested reports will be generated. These reports will contain both tabular statistical data and charts. The output format of these reports will be discussed in Section 3.5.

3.1.2 Use Case Diagram

Our application's UML Use Case diagram is shown in **Figure 3.2** to manifest what users will be able to do with OSM-CAT.

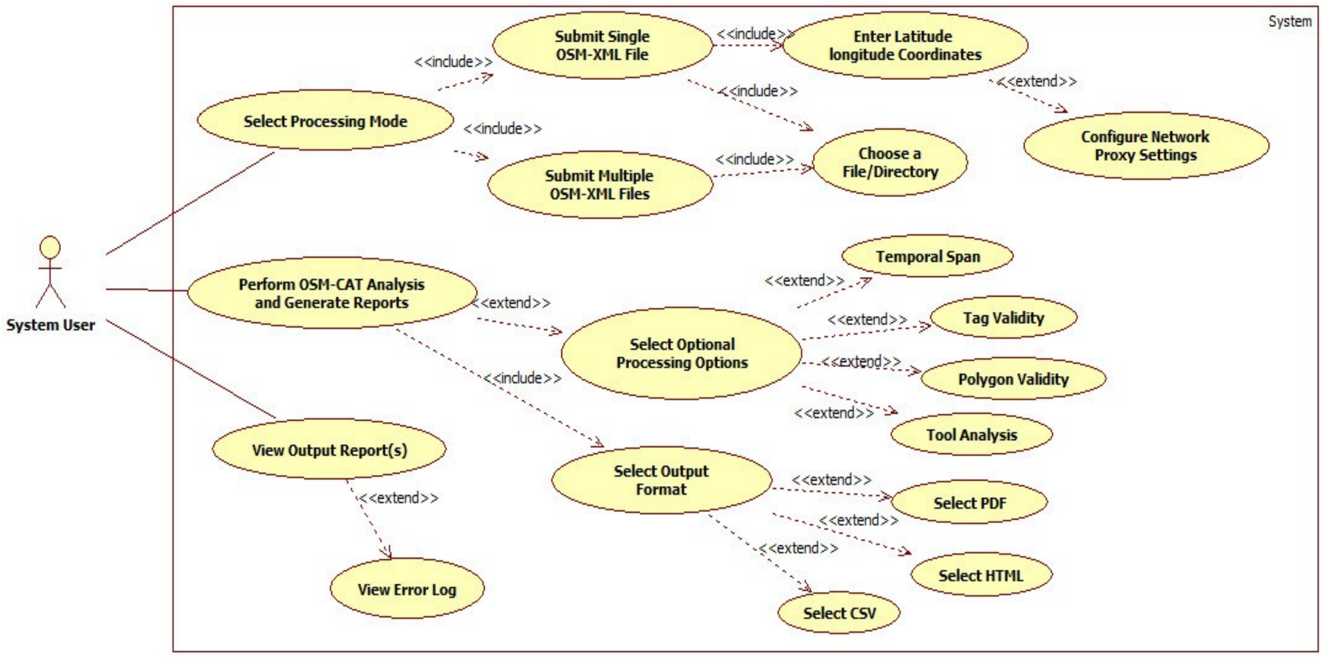


Figure 3.2: OSM-CAT Use Case diagram

These use cases help to convey the functionality provided by OSM-CAT.

3.2 OSM-CAT Functionality Summary

To allow for a deeper understanding of what our software will do, we now present a summary of the entire functionality which will be provided by OSM-CAT.

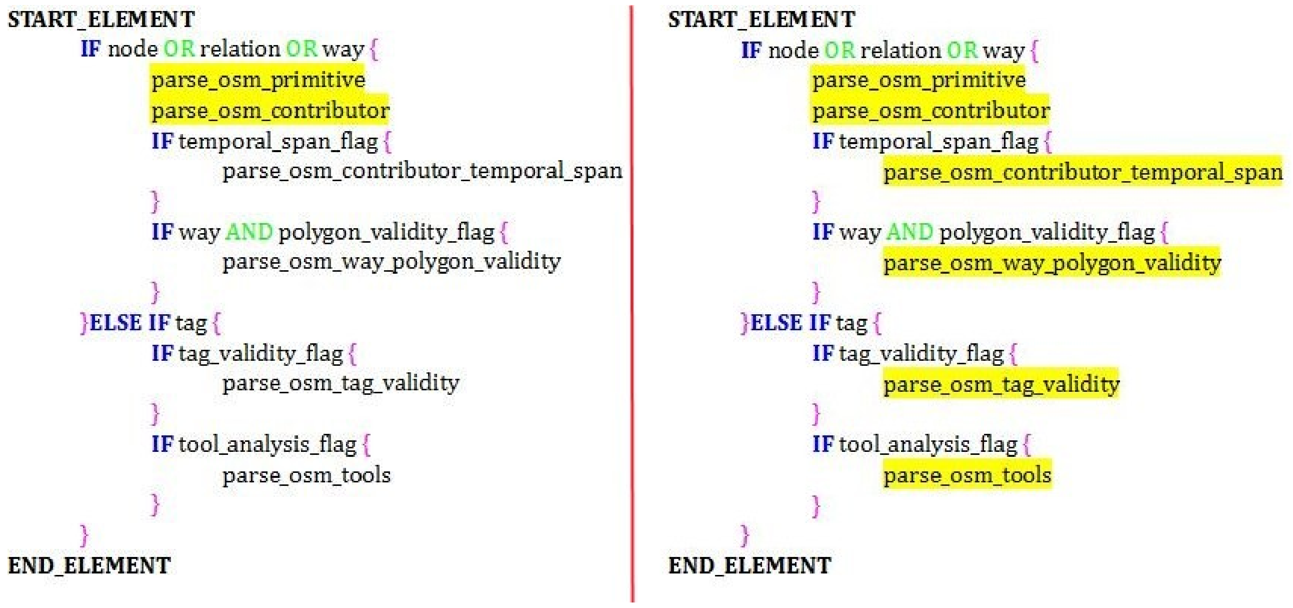
- **OSM Data Summary**: OSM-CAT will provide an OSM primitive data overview including the total number of edits, Nodes, Ways, Relations, and *changesets*. A changeset [36] is a group of edits made within a certain time by one user. A changeset has a maximum capacity (currently 50,000 edits) and maximum lifetime (currently 24 hours).
- **Changeset Analysis**: An analysis of the number of changesets produced by each contributor will be provided. This information will be presented in tabular and graphical formats.
- **Activity in Changesets**: OSM-CAT will provide a summary of the timespan between changesets (longest and shortest) for each contributor.
- **Overall Contributions**: Tabular and graphical summary of the total number of contributions per month since oldest edit in a chosen area will be provided.
- **Individual Contribution Rates**: OSM-CAT will outline the number of edits per month from the top N contributors in both tabular and graphical formats.
- **Map Features and Tag Validity**: Using the OSM Map Features Ontology [37], OSM-CAT will check all tags for objects of a selected set of feature types (See Section 3.3.3.2). This set will correspond to the most frequently occurring objects in OSM according to the TagInfo service [38]. OSM-CAT will provide a summary of the number of valid and invalid assignments to the tag keys for these objects.
- **Map Features and Tag Validity per contributor**: OSM-CAT will provide a summary of the number of valid and invalid tag assignments created by the top N contributors.
- **Creation Tools**: OSM-CAT will provide a pie chart and tabular overview of the most frequently used software tools by the contributors to the current OSM snapshot.
- **Polygon Validity**: Using the JTS library, OSM-CAT will outline any polygons (closed Ways) which are invalid such as those with self-intersections.

3.3 OSM-XML Selective Parser

As discussed in Section 2.2, the efficient processing of arbitrarily-sized, complex OSM-XML files is a very difficult task to perform. As we have shown, the file sizes for even small geographic regions can be relatively large. Such file sizes become a greater concern when we wish to process densely populated, urban areas, where constant user contributions create an abundance of spatial information, causing files sizes to grow rapidly. Complexity is also a concern, and we have shown that even a small selection of OSM-XML files, representing areas of just 1 Km squared, can produce a wide range of File Composition Measures. These issues need to be considered when designing our OSM-XML parser.

3.3.1 Selective Parser Design Overview

In Section 2.1 we discussed conventional methods of processing XML data. We presented two approaches commonly used – the Simple API for XML (SAX), and the Document Object Model (DOM). Based on the findings of our analysis of these parsings APIs, discussed in Section 2.2.1, we designed an OSM-XML parser which builds on the SAX parser API. As outlined, OSM specific XML uses a structure which allows it to capture complex spatial information in a format which can be processed using an XML parser. This volume of data contains information which, depending on the analysis options chosen by the user, is either central to our processing, or is superfluous. Therefore, to facilitate the efficient processing of OSM-XML data, we design a parser which parses only the specific spatial information the user is interested in. This method of selective parsing allows OSM-CAT to parse only that data which is required for a particular analysis. **Figure 3.3** shows an abstract, pseudo-code comparison between basic and full OSM-CAT analysis, comparing the amount of parsing required by each.



```
START_ELEMENT
        IF node OR relation OR way {
                parse_osm_primitive
                parse_osm_contributor
                IF temporal_span_flag {
                        parse_osm_contributor_temporal_span
                }
                IF way AND polygon_validity_flag {
                        parse_osm_way_polygon_validity
                }
        }ELSE IF tag {
                IF tag_validity_flag {
                        parse_osm_tag_validity
                }
                IF tool_analysis_flag {
                        parse_osm_tools
                }
        }
END_ELEMENT
```

```
START_ELEMENT
        IF node OR relation OR way {
                parse_osm_primitive
                parse_osm_contributor
                IF temporal_span_flag {
                        parse_osm_contributor_temporal_span
                }
                IF way AND polygon_validity_flag {
                        parse_osm_way_polygon_validity
                }
        }ELSE IF tag {
                IF tag_validity_flag {
                        parse_osm_tag_validity
                }
                IF tool_analysis_flag {
                        parse_osm_tools
                }
        }
END_ELEMENT
```

Figure 3.3: Comparing basic (left) & full (right) analysis processing

As will be detailed in Section 3.4, our GUI will allow users to select exactly those aspects of the

OSM data which they wish to analyse. If a user wishes to perform a basic analysis, then only basic contributor related information is parsed, as shown on the left in **Figure 3.3**. This is desirable as basic analysis requires far less processing time than to full analysis.

3.3.2 Selective Parser Work-flow

The work-flow of the selective parser we designed in Section 3.3.1 will be as follows:

1. Once the user has selected the file, or files, to be processed, the parser begins at the very start of the OSM-XML file. It is here that the parser extracts the first piece of useful information. Details of the bounding box of the mapped area, specified as latitude and longitude coordinates, are processed and stored by our application.
2. Next, the parser begins to work its way sequentially through the data. It searches for any XML tags. An XML tag is a construct that begins with '<' and ends with '>'. Such tags indicate the start of an OSM-XML element.
3. Once an OSM-XML element has been found, our parser checks to see if it is a required element of the current specified analysis. As previously mentioned, the user can select from a combination of analysis choices, from basic through to full analysis.
4. If the element is required, the parser performs the necessary data extraction by parsing the information contained within the OSM-XML element. For example, if the element represents an OSM primitive, such as a Node, the parser will extract the following information: Contributor ID, Contributor name, *timestamp*, changeset ID, *version*, latitude, longitude. An OSM timestamp captures the time & date of a contributor's edit. A version attribute is used for *optimistic locking,* which is a mechanism OSM uses to control object edits.
5. Once the parser has performed the necessary parsing on an element, it stores the data in memory, in Java storage containers such as ArrayLists, Maps etc.
6. When the parser reaches the end of the file, it passes control back to a resource manager module, which continues with analysis, and generation of the analysis reports.

This work-flow of the OSM-XML parser is abstracted in **Figure 3.4**.

3.3.3 Selective Parsing Options

In our solution, we allow the user to select one, all, or any combination of the following analysis options, which are performed along with the default basic analysis processing:

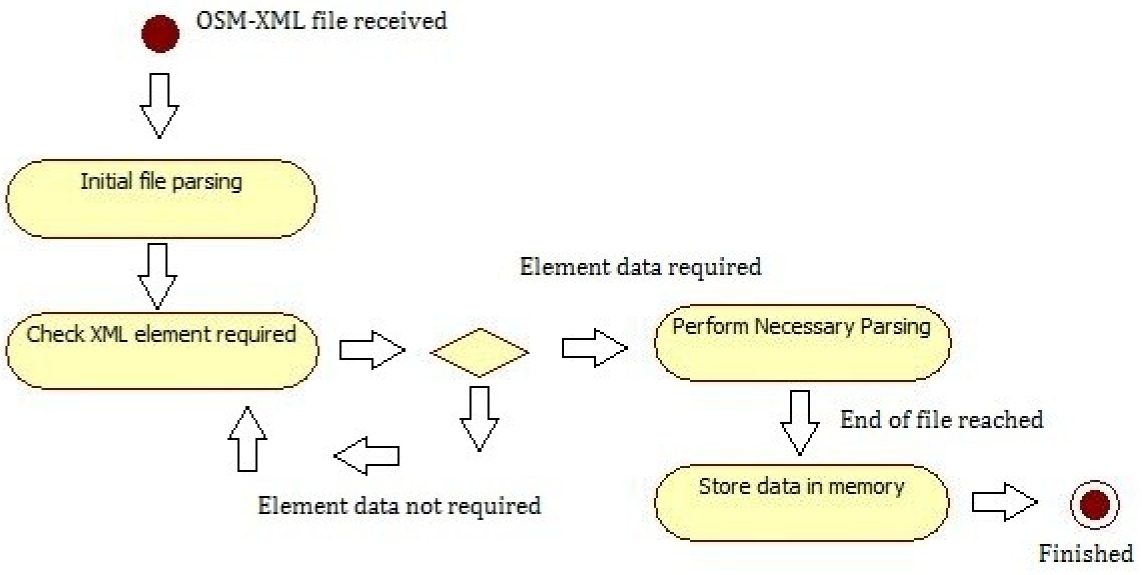


Figure 3.4: Selective parsing work-flow

- Temporal Span
- Tag Validity
- Polygon Validity
- Tool Analysis

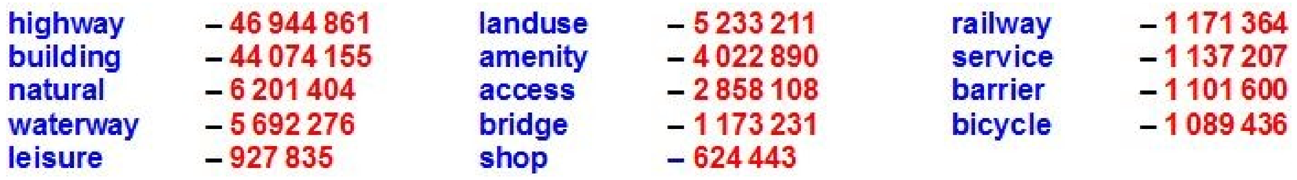
These processing options are further explored in the following sections.

3.3.3.1 Temporal Span

Temporal Span analysis performs processing and analysis relating to the temporal span, or time related, details of the contributors involved in editing a mapped area. This involves collecting information relating to the timestamps of contributor edits. This information will then be used to calculate such statistics as the average number of contributor edits per month, or the very first and last time a contributor edited a map.

3.3.3.2 Tag Validity

Tag Validity performs analysis relating to the tags contained within OSM-XML files. Tags are used in OSM-XML files to represent a variety of information, but of interest to us is tag data relating to the features of a mapped area. Features are spatial elements such as highways or amenities, and these features have tags – a key and value - associated with them, for example *Highway=Motorway.* These values are of particular interest to analysts, as only correctly formatted key-value pairs are considered valid, and can be interpreted by the OSM rendering engine, and other OSM data interpreters. A tag value would be considered invalid if it is not officially specified on the OSM Map Features Ontology page [37]. In our case, we are only interested in a sub-set of the overall available map features, as we feel these represent the majority of features added to maps by contributors. This subset of features was selected by analysing the most frequent tags used by contributors, calculated using TagInfo [38]. **Figure 3.5** shows the sub-set of features we selected, along with a snapshot of the total number of each feature found in the OSM map for the entire planet [39]. Information regarding map features and their corresponding tags will be used by our application to calculate statistical data such as tag correctness and validity.



highway	– 46 944 861	landuse	– 5 233 211	railway	– 1 171 364
building	– 44 074 155	amenity	– 4 022 890	service	– 1 137 207
natural	– 6 201 404	access	– 2 858 108	barrier	– 1 101 600
waterway	– 5 692 276	bridge	– 1 173 231	bicycle	– 1 089 436
leisure	– 927 835	shop	– 624 443		

Figure 3.5: Selection of map features analysed by OSM-CAT

3.3.3.3 Polygon Validity

Polygonal primitives are an important element of spatial data, being used to represent a large variety of map features, such as buildings, lakes, electoral districts, geographic regions etc. Being a widely used element, the validity of polygons in thus an extremely important metric, indicating the quality of the OSM data for any mapped area.

Specifically, a polygon is a figure that is bounded by a closed path, composed of a sequence of straight polylines. A polygon would be considered invalid if the polylines which connect the points of a polygon overlap at any point, in other words, if the polygon self-intersects. **Figure 3.6** depicts valid and invalid polygons.

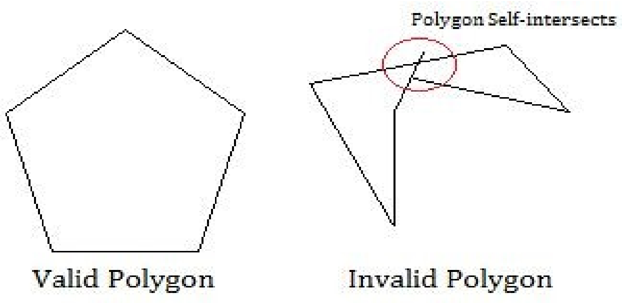


Figure 3.6: Valid and invalid polygons

Polygon data can be used by our application to calculate statistics such as the number of valid/invalid polygons in a given area.

3.3.3.4 Tool Analysis

When contributors edit OSM map data, they do so using one of the numerous editing tools available, such as the JOSM [34] or Potlatch 2 [40] tools. Information regarding what tools are used most frequently, and by contributors in what areas, is of great significance. This data can help highlight issues such as tool usability, tool availability, and language barriers, issues which may affect the adoption of a tool by OSM users. By learning more about these issues, it will be possible for future software developers to create more useful, versatile editing tools.

3.4 Graphical User Interface (GUI)

Usability is a major concern in the design of our application. We want to ensure that OSM-CAT is usable not only by spatial data researchers and professionals, but also by members of the general public, who may have an interest in mapping, and in particular OpenStreetMap. The GUI will be the central, and only, access point to OSM-CAT, through which users can perform every use case outlined in the diagram presented in Section 3.1.2. Therefore, the careful design of our GUI is of critical importance.

Key to this design is simplicity. We want to present the user with an interface that is easily understandable. There should be a minimal learning curve, and the time it takes for users to become proficient in using the application should be minimised. As OSM-CAT will provide the user with a variety of options, such as analysis options and output format options, we need to present these options in a clear and unambiguous manner. In our opinion, the best way to achieve these goals is to use a menu bar interface, which enables users to easily and quickly access the required options. **Figure 3.7** presents our menu bar design prototype.

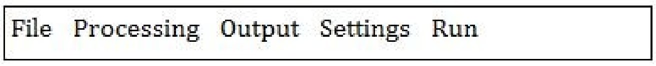


Figure 3.7: GUI menu bar design prototype

Presenting the user with all the necessary options and settings in this way allows for intuitive and quick learning. The user accesses tool features such as file opening, analysis options, and output format options by clicking buttons on the menu bar. **Figure 3.8** shows options displayed to the user in our menu design prototype.

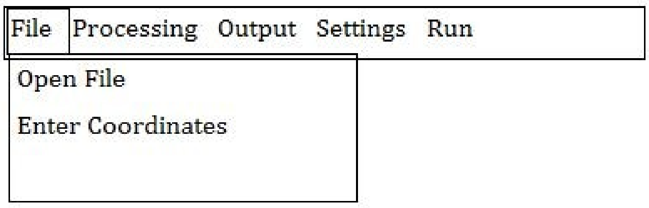


Figure 3.8: GUI menu options design prototype

Feedback to the user regarding processing of the OSM-XML file, such as analysis completion, time taken, and errors encountered, also needs to be considered. Our display is designed to use minimum screen space, while at the same time providing all the necessary options to the user. Therefore, we add a non-intrusive, simple text display area to this design, allowing the user to receive all necessary updates. Our text display area is presented in **Figure 3.9**, along with the complete design prototype of OSM-CAT's GUI.

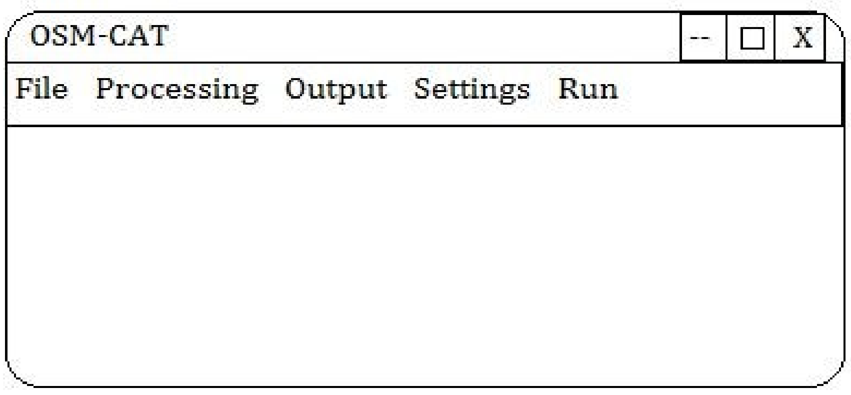


Figure 3.9: Complete OSM-CAT GUI design prototype with text display area

3.5 Report Output Formats

OSM-CAT will allow users to perform analysis of OSM-XML data, and produce detailed reports. It will be possible to generate these reports in three output formats, namely:

- Portable Document Format (PDF)
- HyperText Markup Language (HTML)
- Comma Separated Values (CSV)

The contents of the PDF and HTML output are almost identical, with only small changes to accommodate the constraints of each. The content of the CSV output is slightly different, again due to constraints. The CSV output contains no charts and it takes greater advantage of CSV's tabular format.

To minimise the amount of implementation required to provide different output formats, and promote code reusability and extensibility, we designed the I/O module of our software to accommodate different output formats. This is accomplished by designing an abstract base class, *GenericWriter*, which can be extended and implemented by each of the output format writers. **Figure 3.10** depicts this design.

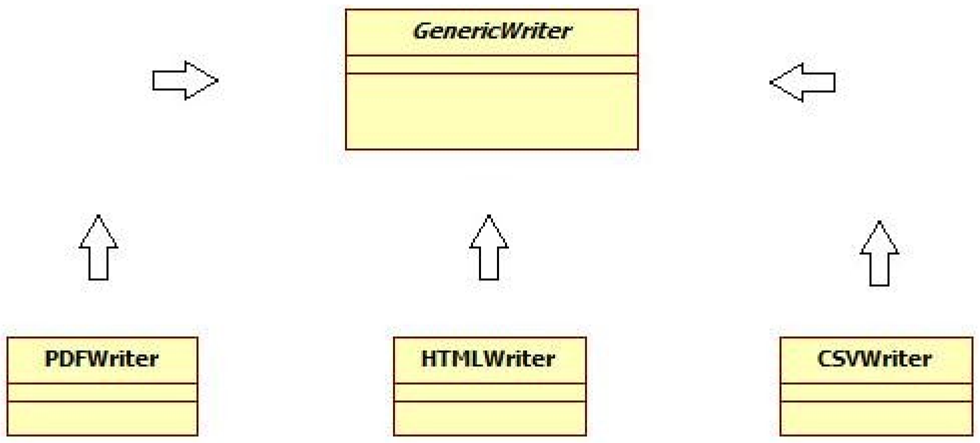


Figure 3.10: Analysis report output writers design

Using this design allows new *Writer* implementations to be added at any stage, ensuring that the software remains extensible. New writers can be 'plugged-in' to the existing software, immediately gaining access to the fields and methods of the abstract *Writer* class. As we intend on releasing our software into the open-source community, and making the code freely available, we feel that these design considerations, allowing for extension and reuse, are of vital importance.

In Section 3, we presented OSM-CAT's design, so that it solves the problem of an efficient, user-friendly contributor analysis application for OSM-XML data. We provided an overview of this design, and presented a summary of the functionality which OSM-CAT will provide. In Section 3.3, we designed the OSM-XML selective parser, which is responsible for reading and processing OSM-XML files. In Section 3.4 our graphical user interface was designed so as to encourage use, and be intuitive to the general user. Finally, in Section 3.5, we discussed the output formats we have chosen for our analysis reports, and presented the design which allows these output formats to be created. Next, in Section 4, we will describe the implementation of our designed application.

4. Implementation

In this section we detail the implementation of OSM-CAT. We take the software design discussed in Section 3, and show how a concrete OSM-XML contributor analysis application is built using existing libraries and frameworks. In Section 4.1, we present an abstract overview of the software modules of OSM-CAT. Section 4.2 details the implementation of the *Input and Output* module of our application, which is responsible for the initial parsing of the OSM-XML, and the creation of reports and charts. In Section 4.3, we discuss the implementation of the *Data* and *MapElements* modules, which are used to store parsed OSM data. Section 4.4 describes the implementation of OSM-CAT's *Graphical User Interface* module, which allows users to interact with our application. Finally, in Section 4.5, we detail the implementation of the *Utilities* module, which provides various functions vital to the operation of OSM-CAT.

4.1 OSM-CAT Software Modules Overview

In **Figure 4.1**, we present a high-level diagram showing the software modules of OSM-CAT, and the dependencies that exist between these modules.

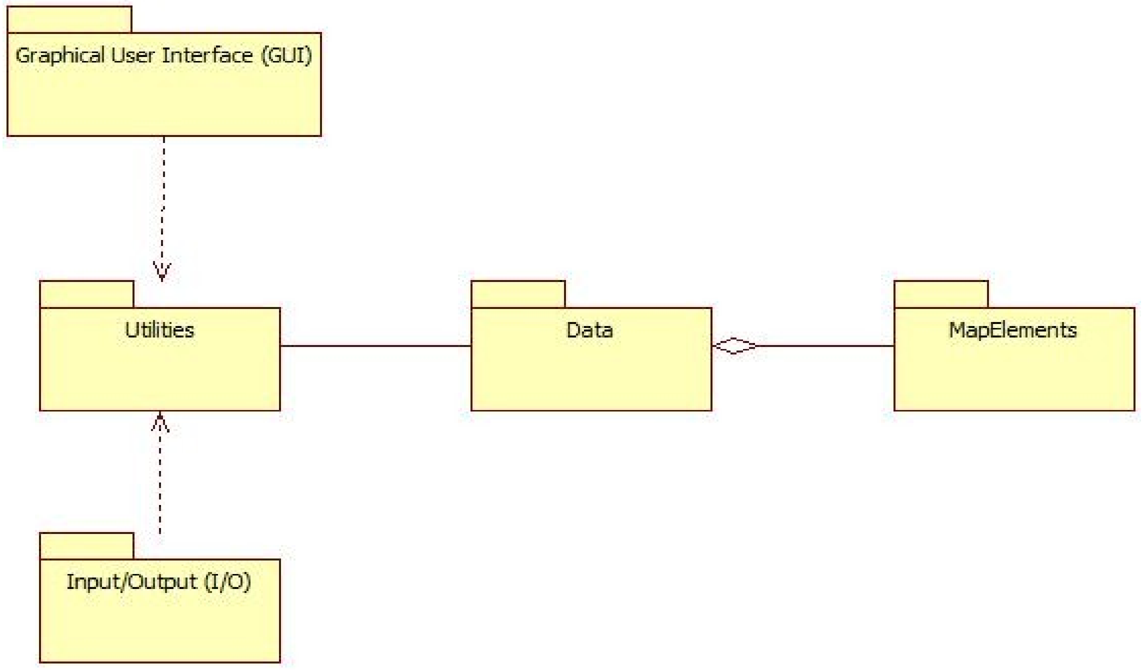


Figure 4.1: OSM-CAT software modules (Java packages)

These modules correspond to Java packages, which are combined to provide OSM-CAT's functionality. Each package contains Java classes and the complete UML Class Diagram for our application is shown in **Appendix A**, **Diagram A.1**. We discuss the implementation of these software modules, and their classes in the sections that follow.

4.2 Input and Output (I/O)

In this section, we illustrate how the *Input and Output* software artefacts are developed. First, we detail the implementation of our OSM-XML parser, based on the SAX parsing API. Then, we demonstrate how iText and the Java HTML Generator are harnessed to produce OSM-CAT analysis reports. We describe the implementation of a chart creator, which draws on the JFreeChart library to create charts based on analysis data. We also describe the modules which create error and invalid feature tag reports. **Figure 4.2** presents an overview of the *I/O* module's classes and their relationships, together with a description of the functionality provided by each. See **Appendix A**, **Diagram A.2** for the full *I/O* module UML Class Diagram.

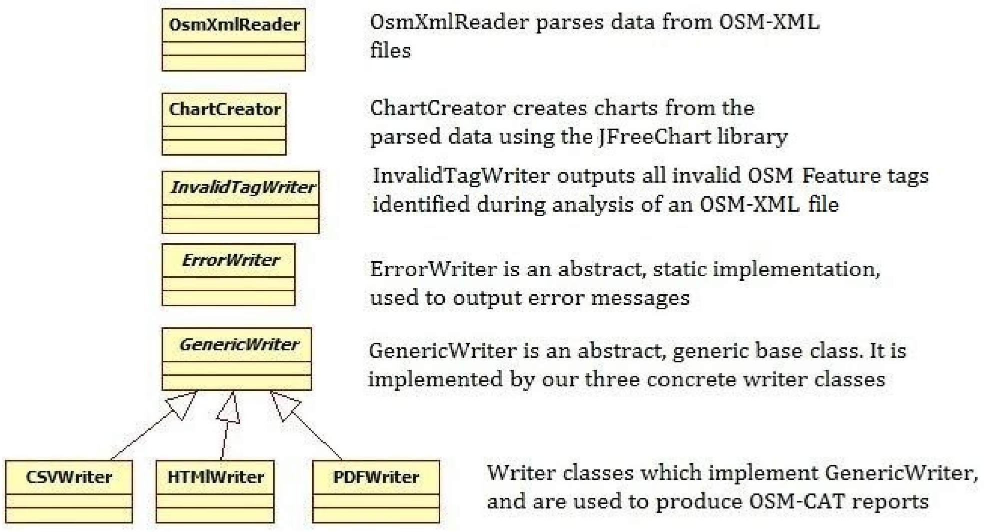
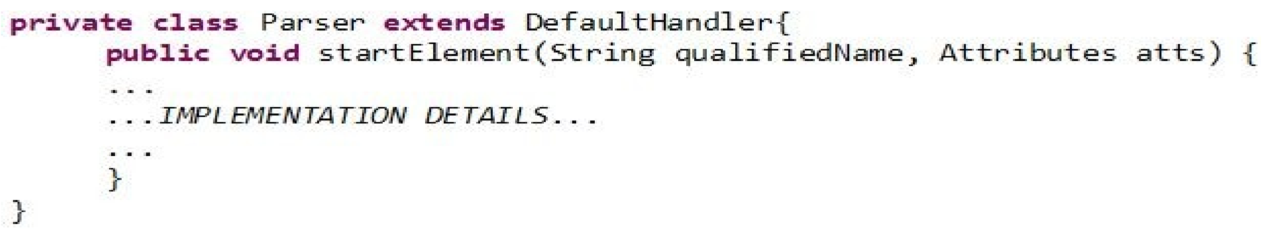

OsmXmlReader
OsmXmlReader parses data from OSM-XML files
ChartCreator
ChartCreator creates charts from the parsed data using the JFreeChart library
InvalidTagWriter
InvalidTagWriter outputs all invalid OSM Feature tags identified during analysis of an OSM-XML file
ErrorWriter
ErrorWriter is an abstract, static implementation, used to output error messages
GenericWriter
GenericWriter is an abstract, generic base class. It is implemented by our three concrete writer classes
CSVWriter
HTMlWriter
PDFWriter
Writer classes which implement GenericWriter, and are used to produce OSM-CAT reports

Figure 4.2: *I/O* module's classes and their functionality

4.2.1 OSM-XML Selective Parsing

Our *OSMXMLReader* parsing class is built on the existing SAX parsing API. This is accomplished by creating an inner class in our *OSMXMLReader* class, *Parser*, which extends the *DefaultHandler* class. This class is the base class for SAX parsing event handlers. Using this underlying base class, we gain access to its methods, specifically the s*tartElement* method, which enables our parser to receive notification of the start of an OSM-XML element. **Figure 4.3** shows an overview of this *Parser* class, and its custom *startElement* method. We override the *startElement* method, defining what we want our parser to do when it encounters the start of an OSM-XML element. Thus, the *startElement* method of our *Parser* class performs the bulk of the OSM-XML processing.



```
private class Parser extends DefaultHandler{
      public void startElement(String qualifiedName, Attributes atts) {
      ...
      ...IMPLEMENTATION DETAILS...
      ...
      }
}
```

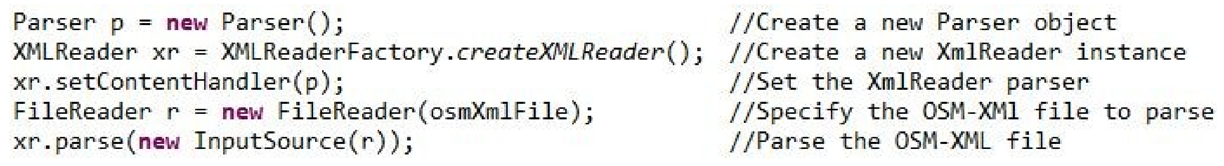
Figure 4.3: ***Parser*** **class and** ***startElement method***

The parameters of this method which we use to extract data from the OSM-XML element are:

- *atts* – List of OSM-XML element attributes. For example: user, uid or changeset
- *qualifiedName* – Starting name of the OSM-XML element, for example Node or Way

These parameters enable OSM-CAT to find and process only the data we are specifically interested in. Our *Parser* class performs selective parsing, processing the minimum amount of data.

When an instance of our *Parser* class is created, it is passed to an *XMLReader* instance. This *XMLReader* instance is the main interface to an XML document, and using our *Parser* class, it performs parsing on an *InputSource* instance. The structure of these parsing calls is shown in **Figure 4.4**.



```
Parser p = new Parser();                                //Create a new Parser object
XMLReader xr = XMLReaderFactory.createXMLReader();      //Create a new XmlReader instance
xr.setContentHandler(p);                                //Set the XmlReader parser
FileReader r = new FileReader(osmXmlFile);              //Specify the OSM-XMl file to parse
xr.parse(new InputSource(r));                           //Parse the OSM-XML file
```

Figure 4.4: Structure of parsing calls

Once our *XMLReader* has reached the end of the document, control is passed back to the *Utilities* module of our software, which we present in Section 4.5.

4.2.2 Report Generation

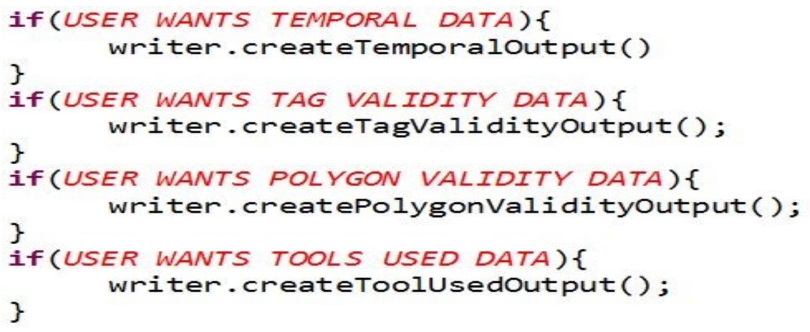
The reports generated by OSM-CAT can be created in three formats: PDF, HTML, and CSV. As shown in **Figure 4.2**, this functionality is implemented using an abstract, generic base class, *GenericWriter*. Our specific output format writers, *PDFWriter*, *HTMLWrier*, and *CSVWriter* are then implemented as concrete sub-classes of *GenericWriter*. This makes use of inheritance[8] and polymorphic method calls[9]. Implementing our design in this way follows good software practice, and provides possibilities for extension and reuse of our software in the future.

8 In object-oriented programming (OOP), inheritance is a method of code reuse which involves establishing sub-types of existing objects.

9 In OOP, polymorphism is a feature which allows different data types to be handled using a uniform interface. Polymorphic method calls refer to the ability to respond to methods calls with type specific behaviour.

Indeed, new output format writers could be implemented and 'plugged-in' to our software. As long as they extend *GenericWriter,* they could be easily substituted anywhere a *GenericWriter* object is being used.

Our *GenericWriter* class's functionality is implemented as methods, enabling dynamic report creation. The modularity of methods ensures that only the analysis report elements specified by the user are created at run-time. This dynamic creation design also utilises logical decision statements in its implementation. **Figure 4.5** shows the pseudo-code structure of OSM-CAT's dynamic report creation. This structure is used for all our writers, as polymorphism allows us to call the correct *GenericWriter* implementation at runtime.



```
if(USER WANTS TEMPORAL DATA){
      writer.createTemporalOutput()
}
if(USER WANTS TAG VALIDITY DATA){
      writer.createTagValidityOutput();
}
if(USER WANTS POLYGON VALIDITY DATA){
      writer.createPolygonValidityOutput();
}
if(USER WANTS TOOLS USED DATA){
      writer.createToolUsedOutput();
}
```

Figure 4.5: Dynamic report creation using decision statements

Our PDF output format is implemented using the open-source iText [32] Java library. This library is used to create and manipulate PDF documents. iText allows us to represent each part of a PDF document using three Java object constructs, namely:

Document element	Corresponding Java use
• *Paragraph*	Paragraph pdfPara = new Paragraph("OSM-CAT");
• *Chapter*	Chapter pdfChapter = new Chapter(pdfParagraph);

• *Document*	Document pdfDoc = new Document();
	Document.add(pdfChapter);

As shown above, iText uses a hierarchical approach, with larger elements, i.e. *Document*, composed of smaller elements, i.e. *Chapter.* Using this approach enables the creation of separate, modular sections of a PDF document. This allows us to build a PDF report dynamically, depending on what analysis options were chosen by the user. Only the chapters which are required are created. **Appendix B**, **Code B.1** shows the creation of a PDF chapter using iText.

Our HTML output format is implemented using the open-source Java HTML Generator [37] package. This package uses a system of *Tag* objects to represent the tags of a standard HTML document. These *Tag* objects are used as follows.

HTML element	Corresponding Java use

• *Tag*	Tag head = new Tag("head");
	Tag body = new Tag("body");
	body.add(head);

Similar to the iText library, the HTML Generator uses a hierarchical structure, with *Tags* representing larger HTML document elements being composed of *Tags* representing smaller elements. The creation of a HTML document section using the HTML Generator is shown in **Appendix B**, **Code B.2**.

Our CSV format requires a comma-separated list of information as output, and is implemented using standard Java libraries, in particular the Java *PrintWriter* class. This *PrintWriter* class allows us to write formatted output, such as a comma-separated list, to a file.

4.2.3 Chart Creation

OSM-CAT creates detailed charts displaying a wide variety of statistical data. These charts are produced using JFreeChart [31], a Java chart library which allows us to create charts in various styles. OSM-CAT creates charts in the following styles:

- Standard Bar Chart
- Stacked Bar Chart
- Box-Plot Chart
- Time-Series Chart
- Pie Chart

The first step in creating a JFreeChart chart is to create the dataset for the chart. The dataset contains the data to be displayed by the chart, and a dataset is created for all charts, from bar charts to pie charts. We then populate this dataset with data from our internal data structures, for example *Nodes* or *Contributors*. These data structures are discussed in Section 4.3. Once the dataset is populated, we pass it as a parameter to a JFreeChart *ChartFactory* method, which uses the data to create the desired chart type. This method of chart creation is shown in **Appendix B**, **Code B.3.** Many aspects of chart creation are shared across all chart styles, allowing us to reuse code, as methods, for tasks such as chart customisation. This follows good software engineering practice.

4.2.4 Invalid Tag Output

InvalidTagWriter is a static implementation which outputs all invalid OSM feature tags found during analysis to an *invalidtagvalues.csv* file. It outputs data in the format *User ID, OSM Feature Tag, Invalid Feature Tag Value*. An example of such output is *99999, Building, Greenhouse*. Here, *Building* is the OSM feature tag, and *Greenhouse* is its invalid feature tag value.

4.2.5 Error Output

Our *ErrorWriter* module is a static implementation used by most modules in our software. If an error occurs during processing, the *ErrorWriter* class allows a corresponding information message to be written out to an automatically created error file, *errorLog.log*. The user is then informed through the GUI that errors have occurred.

We have detailed the implementation of our *I/O* module, which provides functionality for OSM-XML parsing, report generation, and chart creation. In the next section, we detail the implementation of the *Data* and *MapElements* modules.

4.3 Data Storage/Data Structures

The data structures of the *Data* and *MapElements* modules are used to represent the abundance of information that is produced as a result of processing an OSM-XML file. As an OSM-XML file contains representations of geographical spatial elements, we designed and implemented our data structures to correspond to these spatial elements. In **Figure 4.6** we present an overview of both the *Data* and *MapElements* modules' classes and their relationships, together with a description of the functionality provided by each class. The full *Data* and *MapElements* modules' UML Class Diagram is shown in **Appendix A**, **Diagram A.3**.

4.3.1 Map Data

The central module for data storage in our application is the *MapData* class. Once initialised, a *MapData* instance represents the map specified by the latitude and longitude coordinates of the OSM-XML file, which are the bounds of the map. *MapData* builds on the JOSM [34] library,

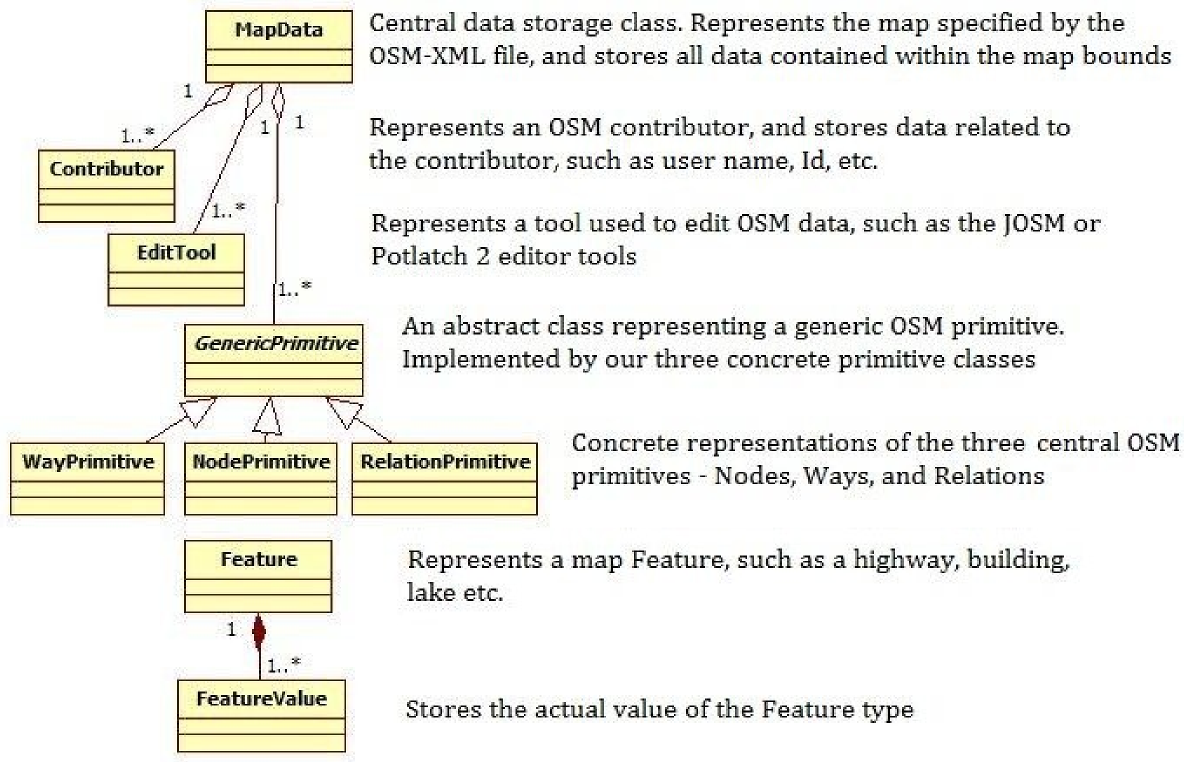

MapData
Central data storage class. Represents the map specified by the OSM-XML file, and stores all data contained within the map bounds
1
1..*
1
1
Contributor
Represents an OSM contributor, and stores data related to the contributor, such as user name, Id, etc.
1..*
EditTool
Represents a tool used to edit OSM data, such as the JOSM or Potlatch 2 editor tools
1..*
GenericPrimitive
An abstract class representing a generic OSM primitive. Implemented by our three concrete primitive classes
WayPrimitive
NodePrimitive
RelationPrimitive
Concrete representations of the three central OSM primitives - Nodes, Ways, and Relations
Feature
Represents a map Feature, such as a highway, building, lake etc.
1
1..*
FeatureValue
Stores the actual value of the Feature type

Figure 4.6: ***Data*** **&** ***MapElements*** **modules' classes and their functionality**

in particular the *Bounds* class, to represent a mapped area, and the *LatLon* class, which represents latitude and longitude coordinates.

The data within map bounds, such as primitive data and contributor data, constitutes the data parsed from an OSM-XML file. Therefore, an instance of the *MapData* class is used to store all of this inter-connected data. In this way, the *MapData* class acts as a central container class for the processed data. Specifically, the *MapData* class stores data relating to the following:

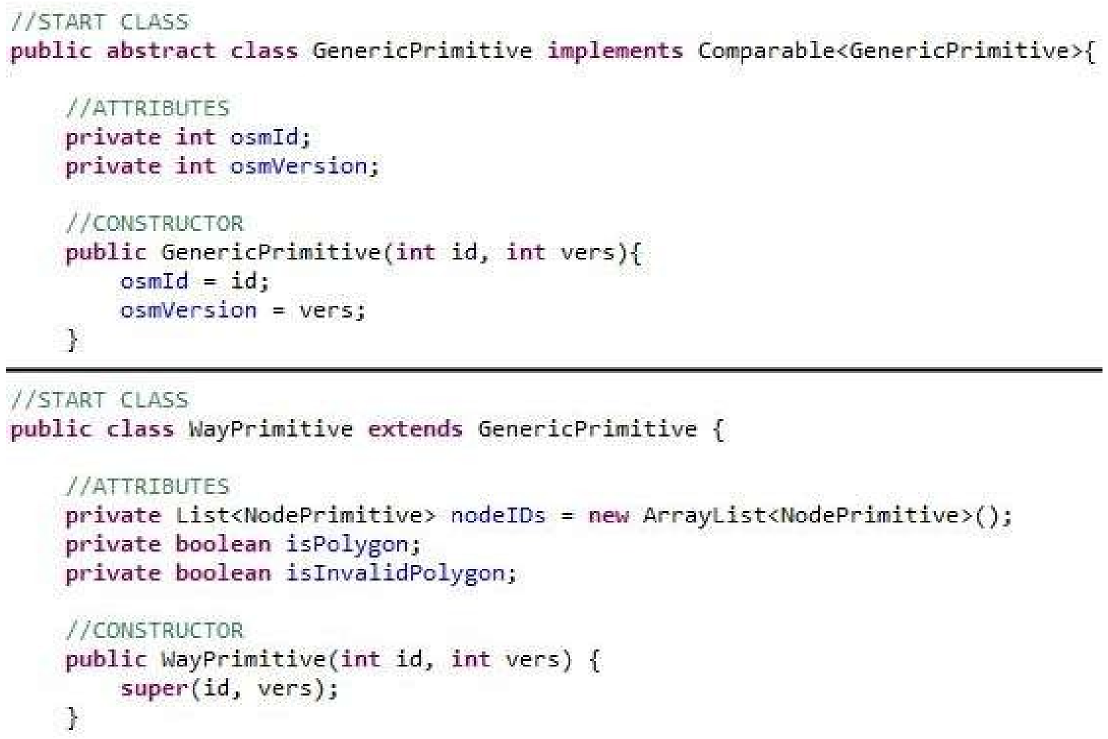
- Map Edits (Nodes, Ways, Relations)
- Map Changeset Edits (Nodes, Ways, Relations)
- Contributor Information
- OSM Primitive Data (Nodes, Ways, Relations)
- Editor Tool Data
- Timestamp Data
- Feature Tag Data

Some of these elements are themselves represented as data structures, and these are discussed further in the following section.

4.3.2 OSM Primitive Data

OSM geometric primitives are of three types: Nodes (points), Ways (polylines), and Relations. These primitives share common attributes, such as their OSM identification number and OSM version number, and so we chose to implement them using inheritance and genericity. This is shown in **Figure 4.6**, where the abstract base class, *GenericPrimitive*, represents an OSM primitive. This generic class and its methods are then implemented by each of the three concrete primitive types. The *NodePrimitive* class is primarily used as a representation of a Node, or point. A *WayPrimitive* represents a Way, or polyline, which are lines connecting Nodes. The Java Topology Suite [35] is used in the implementation of our *WayPrimitive* to check for polygon validity. *RelationPrimitive* represents the relationship between Nodes, Ways, and other Relations, using tags.

Our use of inheritance and genericity allows for dynamic, polymorphic behaviour, where the exact type of a primitive is determined at runtime, and the corresponding implementation of a method is called. This results in cleaner code, reducing the complexity of our software. **Figure 4.7** shows the relationship between the generic base class, *GenericPrimitive*, and one of its implementing sub-classes, *WayPrimitive.*



```
//START CLASS
public abstract class GenericPrimitive implements Comparable<GenericPrimitive>{

    //ATTRIBUTES
    private int osmId;
    private int osmVersion;

    //CONSTRUCTOR
    public GenericPrimitive(int id, int vers){
        osmId = id;
        osmVersion = vers;
    }
```

```
//START CLASS
public class WayPrimitive extends GenericPrimitive {

    //ATTRIBUTES
    private List<NodePrimitive> nodeIDs = new ArrayList<NodePrimitive>();
    private boolean isPolygon;
    private boolean isInvalidPolygon;

    //CONSTRUCTOR
    public WayPrimitive(int id, int vers) {
        super(id, vers);
    }
```

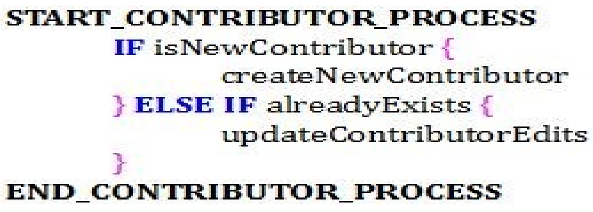
Figure 4.7: Relationship between *GenericPrimitive* and its sub-class *WayPrimitive*

4.3.3 OSM Contributor Data

As OSM-CAT is an OSM contributor analysis application, OSM contributor related information is of great interest. We represent OSM contributors internally as instances of the *Contributor* class. This class has data storage structures corresponding to the contributor-related information required for our analysis. This data includes:

- OSM contributor name – e.g. "Harry Wood"
- OSM contributor ID – e.g. "9999999"
- Number of edits (Nodes, Ways, Relations)
- Number of changeset edits (Nodes, Ways, Relations)
- Temporal span data – e.g. Timestamps, First Use, Last Use etc.
- Feature tagging correctness/validity

When parsing an OSM-XML file, we process the information relating to each *OSMPrimitive* we encounter. Each edit of an OSM primitive is associated with a valid OSM contributor. Thus, if we encounter a user for the first time, we create a corresponding *Contributor* instance, and increment that contributor's edits and changesets. If we encounter a contributor already internally stored as a *Contributor* instance, we search for the *Contributor* and update its data. **Figure 4.8** shows this process of parsing *Contributor* data.



```
START_CONTRIBUTOR_PROCESS
        IF isNewContributor {
                createNewContributor
        } ELSE IF alreadyExists {
                updateContributorEdits
        }
END_CONTRIBUTOR_PROCESS
```

Figure 4.8: *Contributor* parsing process

4.3.4 OSM Feature Data

OSM feature elements represent general, real-world map features, such as highways, buildings, amenities etc. As discussed in Section 3.3.3.2, we selected a subset of the overall OSM feature set for use in our analysis. Each of these 14 features is represented in our application as an instance of the *Feature* class, and each feature has a set of valid *FeatureValues* associated with it. This set of valid feature values is subject to change, as more values are deemed acceptable by OSM. Thus, to future-proof our implementation, we read in the set of valid values from a supplied XML file at runtime. These valid values are then stored in our static *Validtags* class, discussed further in Section 4.5. Our *MapData* data structure stores a reference to a *FeatureUtility* instance, which acts as a bridge between the central *MapData* instance and its required *Feature* and *FeatureValue* data. The *FeatureUtility* class is shown in **Figure 4.16** in Section 4.5.

4.3.5 OSM Editing Tool Data

An instance of the *EditTool* class represents an editing tool, such as Potlatch 2 [40], that was used by a contributor to edit OSM data. A *MapData* instance holds a reference to a list of *EditTools*, which we use as part of our analysis into OSM editing tools. An *EditTool* instance is very inexpensive in terms of memory consumption, as it simply stores attributes representing its name and number of uses.

In this section, we described the implementation of our *Data* and *MapElements* modules, which provide storage structures for all data parsed from an OSM-XML file, such as OSM primitives, contributors, and feature data. In the following section, we describe the implementation of the *Graphical User Interface* module.

4.4 Graphical User Interface (GUI)

As discussed in Section 3.4, the GUI plays a vital role in the use of our application. Indeed, it is the only entry point a user has to OSM-CAT. Its simplicity and ease of use are of great importance, and so we have designed our GUI to be immediately usable, and to require minimal learning time. We have also designed it to constrain precisely what a user can do with our application at each step in its use. In **Figure 4.9** we present an overview of the *GUI* module's classes and their relationships, together with a description of the functionality

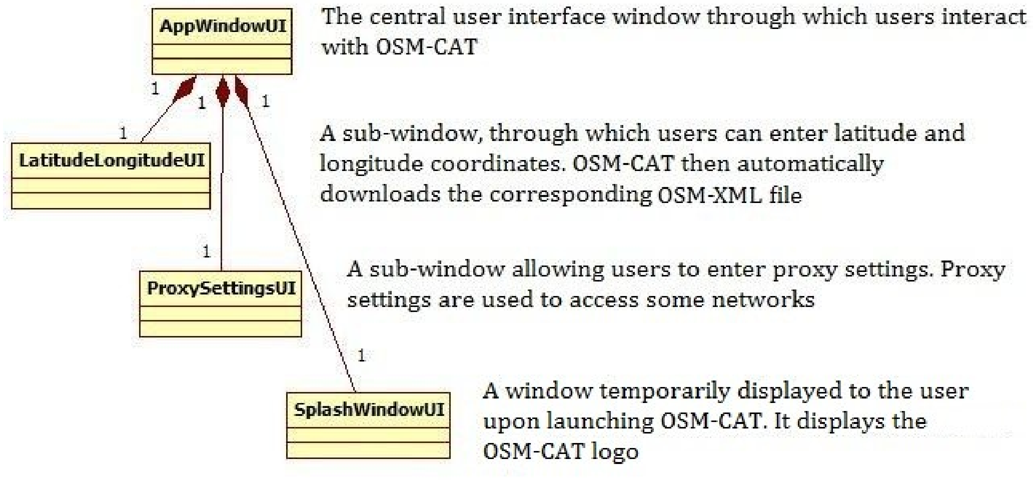

AppWindowUI
The central user interface window through which users interact with OSM-CAT
1
1
1
1
LatitudeLongitudeUI
A sub-window, through which users can enter latitude and longitude coordinates. OSM-CAT then automatically downloads the corresponding OSM-XML file
1
ProxySettingsUI
A sub-window allowing users to enter proxy settings. Proxy settings are used to access some networks
1
SplashWindowUI
A window temporarily displayed to the user upon launching OSM-CAT. It displays the OSM-CAT logo

Figure 4.9: ***GUI*** **module's classes and their functionality**

provided by each class. See **Appendix A**, **Diagram A.4** for the full *GUI* module UML Class Diagram.

4.4.1 GUI Main Window

The single entry point to OSM-CAT is through an instance of the *AppWindow* class. This class represents an application window, which is displayed to the user upon launching OSM-CAT. This application window enables users to perform tasks such as selecting OSM-XML files to analyse, choosing analysis options, and selecting desired output formats. The *AppWindow* instance also performs checks on the OSM-XML file prior to sending the file for processing, such as ensuring the file is a valid OSM-XML file, and that it is less than 100 MB in size. As stated, we place a 100 MB limit on the size of files which OSM-CAT will process.

Based on the design presented in Section 3.4, we produced an application window using the Java Swing and Java AWT libraries. **Figure 4.10** shows this application window.

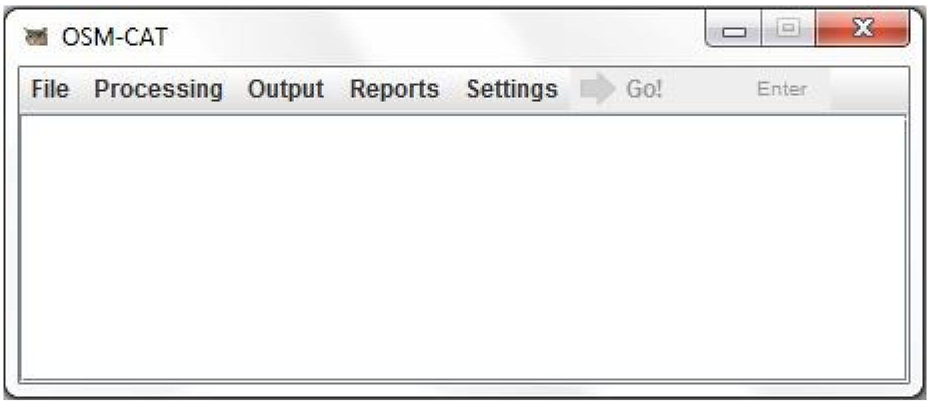
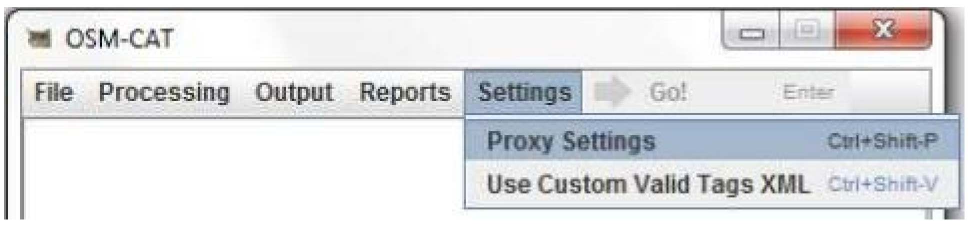


Figure 4.10: OSM-CAT application window

This window is built using a single Swing *JFrame* object, allowing us to achieve fast and efficient execution of the application. Other Swing components are then added, for example a *JScrollPane*, *JMenus*, *JMenuItems*, etc. Being a GUI, our application window needs to respond quickly and correctly to user actions. This event-response relationship is implemented using the Java *awt.event* package, and in particular, the *ActionEvent* class. **Figure 4.11** shows a GUI user action, and the corresponding Java code used to handle that event. Using this method of GUI event handling allows OSM-CAT to effectively and quickly respond to any user initiated events.

As previously mentioned, our GUI design makes use of constraints to make it almost


OSM-CAT
File
Processing
Output
Reports
Settings
Go!
Enter
Proxy Settings Ctrl+Shift-P
Use Custom Valid Tags XML Ctrl+Shift-V

```
if (e.getSource() == proxySettings) {
    final ProxySettingsUI proxyUI = new ProxySettingsUI(frame, log);
    try {
        proxyUI.setDefaultCloseOperation(JDialog.DISPOSE_ON_CLOSE);
        proxyUI.setVisible(true);
    } catch (final HeadlessException ex) {
        log.append("Error opening window. Update Java Environment.");
    }
    proxySetting = proxyUI.getProxy();
}
```

Figure 4.11: GUI user action and corresponding event handling code

impossible for something to go wrong during the use of our tool. We implement these constraints by 'greying out' certain buttons, or options, in the GUI at certain times during the applications use. Interface elements, such as buttons, only become available to a user when they are applicable to the current state of the application. **Figure 4.12** shows a comparison between unavailable and available GUI components. These imposed constraints control what a user can do at each point of OSM-CAT's execution, guiding them through its use. In our opinion, these constraints will assist users in quickly learning how OSM-CAT should be used.

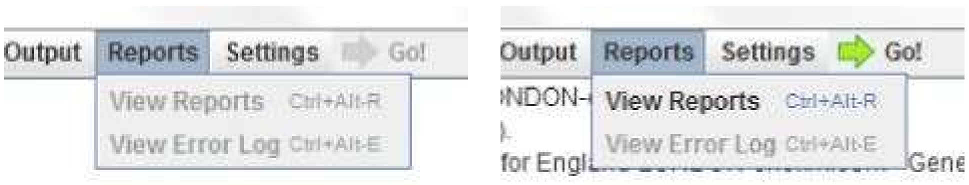


Figure 4.12: GUI imposed constraints – unavailable and available menu components

Our GUI also makes use of keyboard shortcuts, or keystrokes, allowing the user to completely control OSM-CAT using different combinations of keys. These shortcuts are also shown in **Figure 4.12**, where 'Ctrl+Alt+R' is used to 'View Reports'.

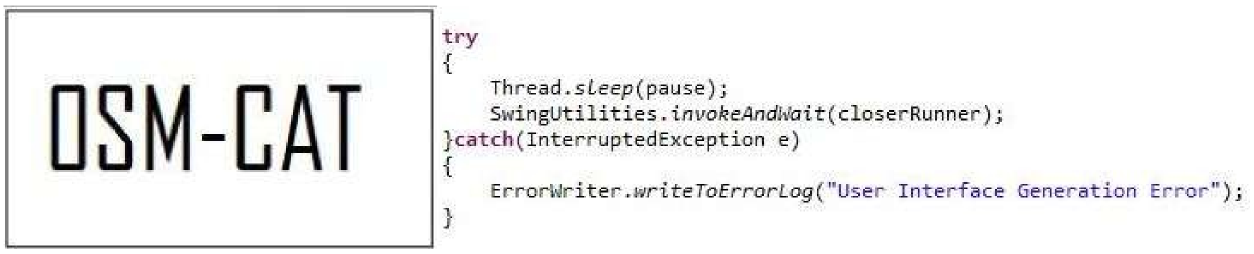
4.4.2 GUI Sub-Windows

Our main application window uses a system of sub-windows to perform a range of user-initiated tasks. The following is a description of these sub-windows, and the functionality they provide.

4.4.2.1 Splash Window

A Splash Window is a window which is displayed upon first launching an application. It is often used to display an image, or text, during the initial loading of an application, which may take a long time. As the loading of OSM-CAT is nearly instantaneous, we primarily use our splash window for aesthetic purposes. However, in the future, as we extend OSM-CAT's capabilities, our splash screen may be used to display information during a loading waiting period.

Our *SplashWindowUI* class is implemented using a *JLabel*, which we use to display the OSM-CAT logo. We use Java threading to execute the display of this *JLabel* as a separate thread. This allows us to display the splash window while not inhibiting the loading of our main application window. It also allows us to specify the amount of time that the splash window is displayed, using thread commands. **Figure 4.13** shows our splash window displaying the OSM-CAT logo, and the corresponding threading code.



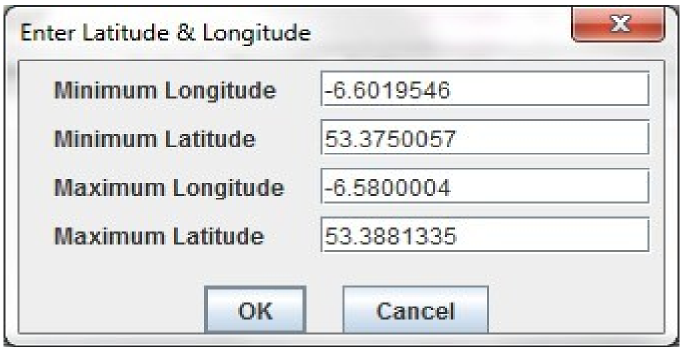
```
try
{
    Thread.sleep(pause);
    SwingUtilities.invokeAndWait(closerRunner);
}catch(InterruptedException e)
{
    ErrorWriter.writeToErrorLog("User Interface Generation Error");
}
```

Figure 4.13: *SplashWindowUI* and corresponding Java threading code

4.4.2.2 Latitude/Longitude Window

As well as allowing users to choose an OSM-XML file, or files, to open from a directory location, OSM-CAT also allows for automatic downloading of a location's OSM-XML data. Users enter the latitude and longitude coordinates of a location, and OSM-CAT automatically connects to the OSM database, and downloads the corresponding OSM-XML data.

Our *LatitudeAndLongitudeUI* class extends the Swing *JDialog* type, and acts as an entry form, allowing the user to enter latitude and longitude coordinates. It then performs necessary checks on the entered information – the bounding box specified by the coordinates must not be too large, latitudes must be valid numbers between -90 and +90, longitudes must be valid


Enter Latitude & Longitude

Minimum Longitude: -6.6019546
Minimum Latitude: 53.3750057
Maximum Longitude: -6.5800004
Maximum Latitude: 53.3881335

Figure 4.14: ***LatitudeAndLongitudeUI*** **window**

numbers between -180 and +180 – before passing the coordinates data to the *AppWindow* instance. The data is then downloaded by the *OSMXMLReader* class. **Figure 4.14** shows a *LatitudeAndLongitudeUI* window instance, which specifies the latitude and longitude coordinates of Maynooth, Ireland [41] by default.

4.4.2.3 Proxy Settings Window

As OSM-CAT uses a network connection when automatically downloading OSM-XML data, we deemed it necessary to enable users to configure the applications proxy settings, allowing OSM-CAT to correctly interact with a proxy server. This is to ensure that the tool is usable in places such as universities, workplaces, libraries, etc., where proxy servers are often in use. A proxy server acts as an intermediary between clients and other servers. The proxy is set by the user and is then used when connecting to the OSM database. **Figure 4.15** shows our *ProxySettingUI* GUI window.

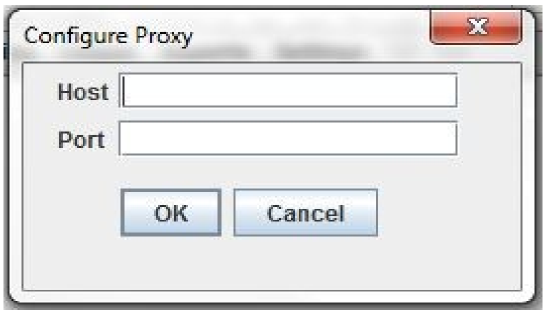


Figure 4.15: ***ProxySettingUI*** **window**

In this section, we detailed the implementation of OSM-CAT's *GUI* module, which enables users to interact with our application. We presented images of most of the GUI's components, and a complete overview of OSM-CAT's GUI is presented in **Appendix D**. In the following section, we detail the implementation of the *Utilities* module.

4.5 Utilities

A vital component of our software implementation is the *Utilities* module, which we discuss in this section. This module contains classes which although central to the successful operation of our application, would not be considered reusable enough to warrant being included in any of the other modules. In **Figure 4.16** we present an overview of the *Utilities* module's classes and their relationships, together with a description of the functionality provided by each class. The full *Utilities* module UML Class Diagram is shown in **Appendix A**, **Diagram A.5**.

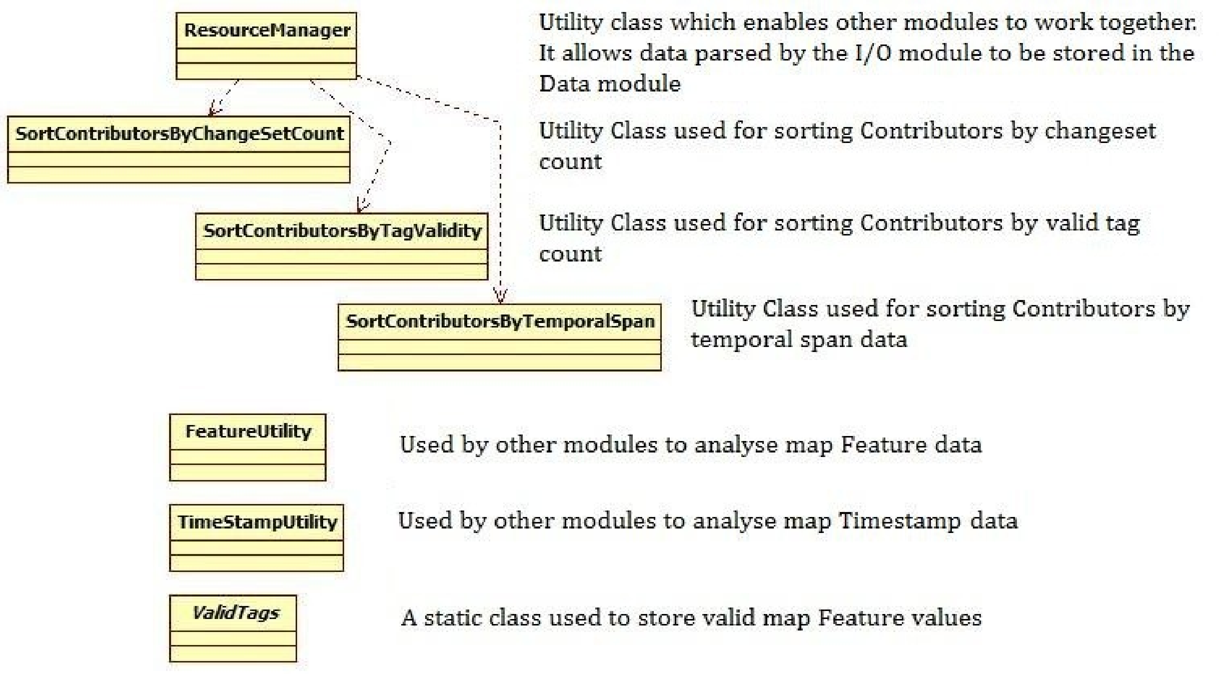

ResourceManager
Utility class which enables other modules to work together. It allows data parsed by the I/O module to be stored in the Data module
SortContributorsByChangeSetCount
Utility Class used for sorting Contributors by changeset count
SortContributorsByTagValidity
Utility Class used for sorting Contributors by valid tag count
SortContributorsByTemporalSpan
Utility Class used for sorting Contributors by temporal span data
FeatureUtility
Used by other modules to analyse map Feature data
TimeStampUtility
Used by other modules to analyse map Timestamp data
ValidTags
A static class used to store valid map Feature values

Figure 4.16: *Utilities* module's classes and their functionality

4.5.1 Resource Management

The *ResourceManager* class enables other software modules to work together. It contains references to two important class instances; *MapData* and *OSMXMLReader*. By linking these two resources, data parsed using the *OSMXMLReader* can be stored in *MapData*, avoiding unnecessary binding between software modules. This ensures that two of the primary software engineering principles are adhered to, namely, low coupling[10] and high cohesion[11]. The *ResourceManager* class is designed to be used specifically in our application, rather than as a particularly reusable component.

10 Coupling is the degree to which software modules rely on each other. Low coupling is desired.

11 Cohesion is a measure of the relationships between functions in a software module. High cohesion is desired.

4.5.2 Timestamp Utilities

Timestamps are used throughout our analysis to calculate statistics regarding both individual contributor temporal data, and overall map temporal data. As numerous temporal-related methods are called by various different modules, such as the *PDFWriter* and *ChartCreator* modules, we designed a *TimeStampUtility* class to implement our methods. This allows these methods to be utilised by all classes, without the need to increase the coupling of components.

4.5.3 Feature Utilities

In a similar way to our *TimeStampUtility* class, our *FeatureUtility* class is used by various other software modules. Its central role is to connect *Feature* instances with modules such as our *MapData* and *GenericWriter* implementations. The *FeatureUtility* class stores references to the sub-set of *Features* that we discussed in Section 4.3.4, and performs such functions as returning the valid and invalid tag value count for a particular feature.

The static *ValidTags* class is used solely as a storage and reference class for valid *Feature* values. It reads in valid tags values from an XML file, and is used by other classes to check the validity of parsed tag values. We provide a default XML file, packaged with OSM-CAT, but users can also use the GUI to select and specify their own XML file. Thus, if a user wants to add a new valid feature, or add new valid values for a feature, it's possible, without requiring any changes to the underlying software implementation. **Appendix C**, **Figure C.1** shows a sub-section of our default valid tag XML file.

4.5.4 Comparator Utilities

Much of our processing involves sorting various data structures by different metrics. In particular, we perform many different sorting operations on the *Contributor* data structures referenced in the *MapData* instance. Our *ComparatorsUtility* file contains three separate classes, each of which implements the Java *Comparator* interface. These three classes are used by other modules to sort the *Contributor* data structures according to different metrics. **Appendix B**, **Code B.4** shows a *Comparator* implementation along with its use in another module.

In Section 4, we detailed the implementation OSM-CAT to meet the design specified in Section

3. In Section 4.2, we built the *I/O* module, with the assistance of iText, the Java HTML Generator, and JFreeChart. In Section 4.3, we drew on the JOSM library to implement our *Data* and *MapElements* modules, and in Section 4.3.2, we implemented our OSM primitive modules with the assistance of the Java Topology Suite. In Section 4.4, we constructed the OSM-CAT *GUI* module using the Java Swing library, and we presented our main application user interface. Finally, in Section 4.5, we demonstrated how our *Utility* module is implemented, and how it is used by various other modules. Next, in Section 5, we will conduct experimental analysis to assess the performance of OSM-CAT.

5. Experimental Analysis

In this section we present our experimental analysis of OSM-CAT. In Section 5.1, we provide an overview of the experimental setup for this analysis. We discuss the process of selecting OSM-XML files for our analysis in Section 5.1.2, including the criteria for selecting these files, the characteristics of the selected files, and the geographic locations which they represent. Section 5.1.3 presents the experimental procedure we followed in our analysis. In Section 5.2, the results from our experimental analysis of OSM-CAT are detailed. With our proposed goals in mind, our analysis focuses on OSM-CAT's efficiency of execution in relation to arbitrarily-sized, and arbitrarily-complex, files, and on the correctness and completeness of the produced analysis data.

5.1 Experimental Setup

This section provides an overview of the experimental setup used for the analysis presented in Section 5.2.

5.1.1 Experimental Environment

All aspects of our experimental analysis were carried out under the same conditions, and in the same environment. In this case, the experimental environment chosen was a single desktop machine running a quad core Intel i5 2.67GHz processor with 6GB of RAM, and the operating system used was Ubuntu 11.10.

5.1.2 OSM-XML File Test Corpus Selection

One of our central design goals was to design an application which efficiently processed OSM-XML files of arbitrary size and complexity. Therefore, in our opinion, to adequately analyse the performance of OSM-CAT, it was necessary to select a test corpus of OSM-XML files which meet four key criteria, specifically:

1. The OSM-XML files selected must be of varying, mostly unique, file sizes.
2. The OSM-XML files must provide varying levels of complexity.
3. The OSM-XML files selected should represent geographically diverse areas.
4. The OSM-XML files must represent OSM maps with varying levels of user contributions.

To satisfy these four criteria, we first decide on a list of mapped European cities [42], shown in **Figure 5.1**. We then take the latitude and longitude of each city from their corresponding Wikipedia page (we take this as being an authoritative source). Using the OSM History Splitter tool[12] [43], we generate bounding boxes of 1, 2, and 3 kilometre square grids, for each of the selected European cities. This provides three OSM-XML files for each of our selected cities, with areas of 1, 2 and 3 Km squared respectively. The idea behind our selection of locations, and sizes, is two fold:

1. To check the efficiency of OSM-CAT as we provide it with larger datasets of varying complexity. These datasets represent geographically diverse areas, with varying levels of user contributions.
2. To measure OSM-CAT's ability to produce correct, valid analysis data.

We will examine OSM-CAT's performance in relation to both file size and file complexity. In our opinion, an OSM-XML file's complexity is determined by its contents, or composition, and so we will use our *File Composition Measure (FCM)* as a measure of a file's complexity. As with our problem analysis, shown in Section 2.2.2, we measure the FCM of an OSM-XML file by selecting a fixed ceiling value, and then expressing the complexity of the file against it. As Paris, France is the largest of our selected OSM-XML files, for 1, 2 and 3 Km square grids, we base the selection of our fixed values on this location. We select a fixed value for both Nodes and Ways, for each of our three grid sizes. Our selected values are shown in **Table 5.1**.

Square Grid Size (Km)	Nodes (N_c)	Ways (W_c)
1	37,000	5,000
2	150,000	20,000
3	310,000	40,000

Table 5.1: Selected fixed ceiling values for each of our grid sizes. These fixed values are used to calculate the FCMs of our OSM-XML test corpus files

Using these fixed values for N_c and W_c, we can now calculate N/N_c, and W/W_c for each of our test corpus OSM-XML files in each square grid size, where N and W are the number of Nodes and Ways in each file. The FCM of each file is then calculated by adding the two results. We will use this FCM, along with a file's size, when we analyse OSM-CAT and draw conclusions on its performance.

12 A Tool used to split full history Planet-Dumps into smaller extracts based on Bounding-Boxes. A Planet-Dump consists of all the OSM data for the entire planet in one file.

We have selected our test corpus of OSM-XML files to satisfy the criteria previously discussed. Firstly, the European cities we selected provide a wide variety of file sizes, and it is highly unlikely that any two of the corresponding OSM-XML files have identical file sizes. Secondly, we constructed bounding boxes of three distinct grid sizes for each European city selected. By doing so, we provide OSM-CAT with files which vary greatly in the number of Nodes and Ways, and consequently their FCM. The number of Nodes and Ways contained in a file of any city should significantly increase as the area of the bounding box increases. Thirdly, we selected locations which we believe represent geographically diverse areas. The population count across our selected 58 cities varies greatly, and consequently so too does the geography of each city. Finally, due to the difference in population across the selected cities, the three different selected grid sizes of each city – 1, 2, 3 Km - and the continuing increase in OSM usage [12], the OSM-XML files selected should exhibit varying levels of user contribution. For a full list of the selected cities, along with file sizes, FCMs, and processing times for each of the three grid sizes of each city, see **Table 5.2** in Section 5.2.

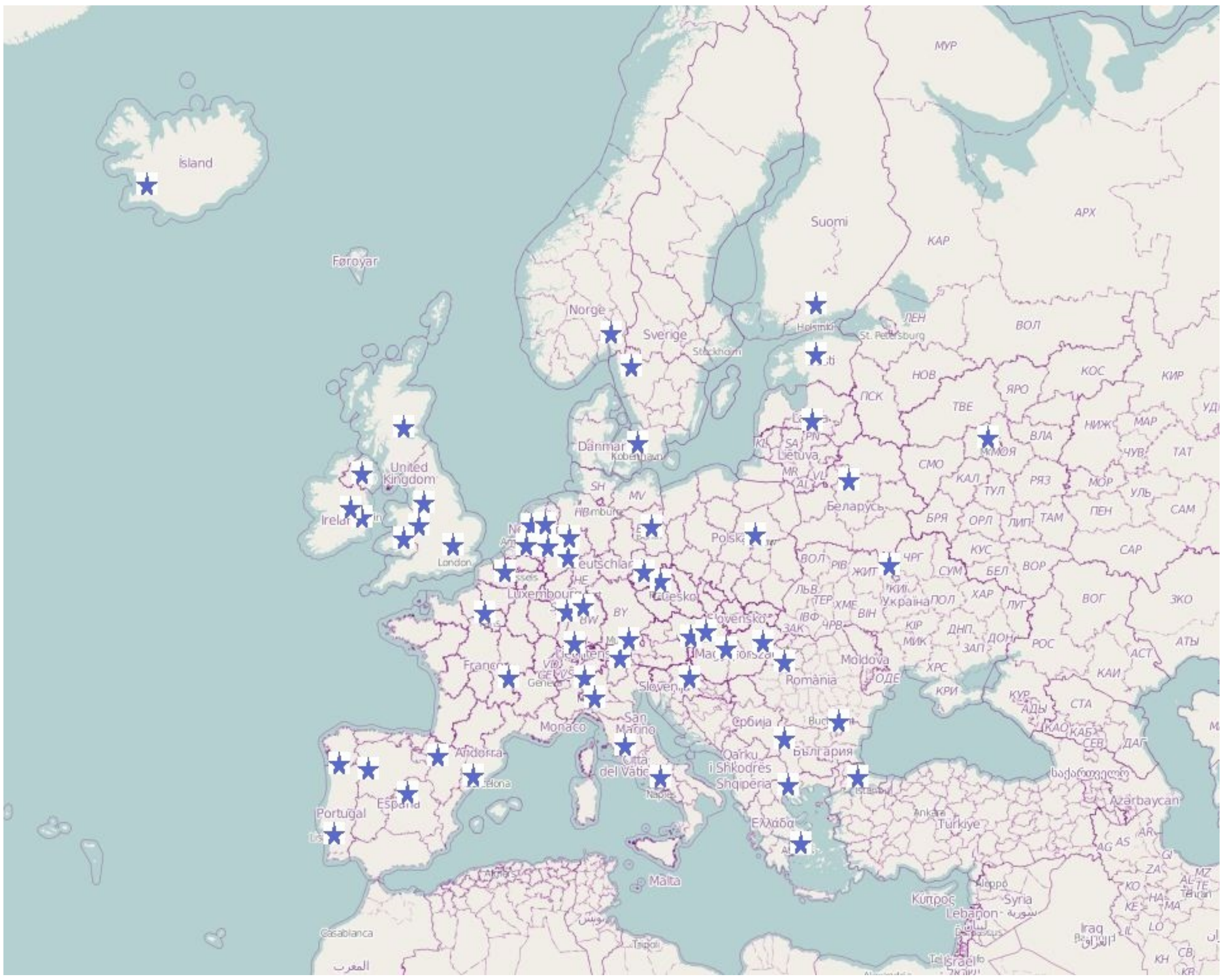


Figure 5.1: Our 58 selected European cities. Each city is used as the source of 3 individual OSM-XML files. The total set of 174 OSM-XML files constitutes our test corpus

5.1.3 Experimental Procedure

We performed full OSM-CAT analysis on each of our selected 58 cities, for each square grid size. Each city was captured with square grid sizes of 1, 2, and 3 Km squared, meaning full analysis was performed on 174 individual OSM-XML files. A full analysis involves processing an OSM-XML file using all analysis options available, which are:

- Basic (default, always performed)
- Temporal Span
- Tag Validity
- Polygon Validity
- Tool Analysis

A single output format, PDF, was chosen, as we believe that most users would require only one of the three formats for any one analysis. The resulting output produced by our full analysis setup is as follows:

- Directory containing set of 13 Chart JPEG[13] image files
- OSM-CAT Report PDF file
- Invalid Tags Report CSV file
- Error Report LOG file

For consistency, the same processing options and output format was used across all 174 files.

5.2 Results

We collected the data relating to a full OSM-CAT analysis of each of the 174 OSM-XML files. **Table 5.2** shows the results of our experimental analysis. All times shown are in seconds.

Country	City	Size 1KM	Size 2KM	Size 3KM	FCM 1KM	FCM 2KM	FCM 3KM	Time 1KM	Time 2KM	Time 3KM
Austria	Innsbruck	3.4 MB	10.5 MB	17.4 MB	0.9777	0.7248	0.5847	13.493	67.442	165.058
Austria	Wien	3.9 MB	10.5 MB	21.5 MB	0.8568	0.6248	0.6309	11.660	56.455	215.669
Belarus	Minsk	1.3 MB	5.2 MB	10.4 MB	0.3349	0.3367	0.3383	4.033	19.590	60.570
Belgium	Brussels	1.9 MB	4.8 MB	9.8 MB	0.5030	0.3261	0.3574	5.548	16.776	56.988
Bulgaria	Sofia	310.6 KB	1.4 MB	2.9 MB	0.0908	0.1022	0.1059	2.067	4.180	8.578
Croatia	Zagreb	761.5 KB	2.3 MB	3.9 MB	0.2161	0.1632	0.1398	2.798	6.727	12.796
Czech Republic	Prague	5 MB	15.7 MB	31.3 MB	1.1261	0.9091	0.8994	17.417	120.256	428.184
Denmark	Copenhagen	5.6 MB	19.1 MB	32.9 MB	0.8482	0.6486	0.5134	11.974	68.712	155.715
England	London	1.9 MB	8.2 MB	18.3 MB	0.4946	0.5407	0.6030	5.323	37.973	157.685
England	Manchester	522 KB	2.3 MB	4.7 MB	0.1464	0.1648	0.1689	2.336	6.398	15.450
Estonia	Tallin	2.8 MB	8.3 MB	14.1 MB	0.7491	0.5593	0.4735	9.269	46.892	117.615
Finland	Helsinki	5.1 MB	14.9 MB	23 MB	1.2600	0.9445	0.7212	15.769	97.225	221.899

13 JPEG (Joint Photographic Experts Group) is a method of image compression for image file formats.

Country	City	Size 1KM	Size 2KM	Size 3KM	FCM 1KM	FCM 2KM	FCM 3KM	Time 1KM	Time 2KM	Time 3KM
France	Lyon	5.1 MB	20.6 MB	38.3 MB	1.2690	1.2839	1.1909	20.686	230.951	749.273
France	Paris	8.8 MB	35.2 MB	71.9 MB	1.8723	1.8561	1.8748	43.186	543.920	2393.165
Germany	Berlin	1.5 MB	7.5 MB	17.3 MB	0.4363	0.4945	0.5699	4.795	34.692	152.680
Germany	Freiberg	1.3 MB	3.6 MB	5.8 MB	0.3582	0.2565	0.2011	3.624	11.240	23.099
Germany	Gelsenkirchen	813.2 KB	3.2 MB	6.4 MB	0.1921	0.1932	0.1911	2.752	9.851	26.262
Germany	Heidelberg	1.7 MB	4.1 MB	8.8 MB	0.4230	0.2542	0.2845	5.224	14.219	48.501
Germany	Karlsruhe	1.6 MB	9.1 MB	21.8 MB	0.4671	0.6202	0.7454	5.499	56.327	267.210
Germany	Meunster	997.6 KB	3.9 MB	7.5 MB	0.2628	0.2690	0.2648	3.098	11.900	32.503
Germany	Munich	3.5 MB	10.3 MB	20.2 MB	0.8931	0.6823	0.6729	10.366	55.381	188.891
Germany	Ruhr	3.9 MB	13.4 MB	28.8 MB	1.1000	0.9042	0.9680	12.798	87.563	366.460
Germany	Stuttgart	3 MB	10.1 MB	20.4 MB	0.7924	0.7245	0.7420	8.615	57.002	208.224
Greece	Athens	499.3 KB	2.1 MB	4 MB	0.1523	0.1567	0.1454	2.281	6.417	14.072
Greece	Thessaloniki	247.7 KB	794.9 KB	1.6 MB	0.0600	0.0524	0.0548	1.884	2.902	4.765
Hungary	Budapest	817.7 KB	2.2 MB	4.6 MB	0.2151	0.1448	0.1494	2.826	6.170	15.291
Hungary	Debrecen	115.8 KB	535.1 KB	1010.9 KB	0.0357	0.0400	0.0372	1.680	2.332	3.406
Iceland	Reykjavik	1.8 MB	11.6 MB	19.6 MB	0.4113	0.6792	0.5622	5.134	62.799	158.789
Ireland	Dublin	1 MB	3.8 MB	7.8 MB	0.2672	0.2783	0.2961	3.057	11.029	32.970
Ireland	Maynooth	2.5 MB	3.6 MB	4.1 MB	0.7205	0.2521	0.1392	7.815	12.383	14.590
Italia	Milano	2.1 MB	7 MB	13.6 MB	0.5839	0.4532	0.4383	6.552	33.647	106.633
Italia	Piacenza	338.4 KB	950.1 KB	1.4 MB	0.1036	0.0701	0.0532	1.974	3.249	4.568
Italia	Roma	1.1 MB	8.8 MB	17.4 MB	0.2821	0.5183	0.5276	3.326	51.082	168.381
Italy	Naples	237.7 KB	703.4 KB	1.4 MB	0.0667	0.0482	0.0503	1.969	2.789	4.533
Latvia	Riga	1.4 MB	3.8 MB	6 MB	0.3809	0.2528	0.1962	4.191	12.576	23.951
Northern Ireland	Belfast	517.2 KB	1.4 MB	2.4 MB	0.1597	0.1028	0.0911	2.392	4.323	7.588
Norway	Oslo	2.8 MB	8.5 MB	19 MB	0.8044	0.5819	0.6364	9.150	44.885	186.294
Poland	Warsaw	1.6 MB	4.6 MB	8.1 MB	0.4020	0.2956	0.2641	4.590	15.082	37.356
Portugal	Braga	285.1 KB	967.3 KB	1.8 MB	0.0786	0.0691	0.0648	1.978	3.328	5.622
Portugal	Lisbon	3 MB	6 MB	8.6 MB	1.0134	0.4941	0.3447	10.992	30.733	56.291
Romania	Bucharest	754.9 KB	1.8 MB	3.4 MB	0.1941	0.1279	0.1179	2.833	5.449	10.756
Romania	Cluj	514.6 KB	1.1 MB	2.2 MB	0.1237	0.0705	0.0693	2.249	3.452	6.095
Russia	Moscow	2.6 MB	9.1 MB	17.9 MB	0.7208	0.5936	0.5825	7.814	47.090	154.012
Scotland	Edinburgh	359.3 KB	3 MB	7.3 MB	0.0968	0.2023	0.2504	2.186	10.057	34.710
Slovakia	Bratislava	4.2 MB	12 MB	21.8 MB	1.0060	0.7424	0.6961	15.975	86.982	256.446
Spain	Barcelona	857.5 KB	2.5 MB	4.1 MB	0.2606	0.1756	0.1443	3.178	8.323	15.887
Spain	Madrid	378.4 KB	1.4 MB	3.5 MB	0.1029	0.0947	0.1191	2.065	4.014	10.100
Spain	Salamanca	5.3 MB	15.3 MB	21 MB	1.8759	1.3500	0.8863	17.979	100.076	178.914
Spain	Zaragoza	773.1 KB	2.4 MB	4.5 MB	0.2454	0.1910	0.1737	2.805	6.929	15.522
Sweeden	Gothenburg	1.4 MB	5.8 MB	11.2 MB	0.4016	0.3991	0.3801	4.516	25.024	77.196
Switzerland	Zurich	2.7 MB	9.2 MB	17.4 MB	0.8234	0.6811	0.6245	8.454	49.855	152.974
TheNetherlands	Amsterdam	2.2 MB	6.9 MB	14.4 MB	0.6358	0.4978	0.5042	6.038	28.107	110.876
TheNetherlands	Eindhoven	1.9 MB	7.6 MB	16 MB	0.5024	0.5241	0.5407	5.872	39.225	142.683
TheNetherlands	Utrech	2.1 MB	7.8 MB	18.1 MB	0.5502	0.5280	0.6134	6.042	36.679	165.778
Turkey	Istanbul	576.5 KB	1.8 MB	2.5 MB	0.1637	0.1354	0.0912	2.525	5.783	8.124
Ukraine	Kiev	1 MB	3.3 MB	5.7 MB	0.2838	0.2307	0.1961	3.480	11.340	24.592
United Kingdom	Birmingham	2.8 MB	8.7 MB	17.6 MB	0.7101	0.5500	0.5631	7.670	40.944	150.667

Country	City	Size 1KM	Size 2KM	Size 3KM	FCM 1KM	FCM 2KM	FCM 3KM	Time 1KM	Time 2KM	Time 3KM
Wales	Cardiff	1.7 MB	6.1 MB	10.9 MB	0.4261	0.3946	0.3554	5.0390	25.561	67.275

Table 5.2: Experimental analysis results for 58 selected European cities

Examining **Table 5.2,** we first note that the size of an OSM-XML file greatly increases as the size of the mapped area increases. Debrecen, Hungary which has the smallest 1 Km file size of 115.8 KB, increases in file size to 535.1 KB for 2 Km, and to 1010.9 KB for 3 Km, percentage increases of 362% and 89% respectively. The city with the largest 1 Km file size of 8.8 MB, Paris, France, increases in file size to 35.2 MB, and to the very large file size of 71.9 MB for 3 Km, percentage increases of 300% and 104% respectively.

However, interestingly, the FCM of an OSM-XML file either decreases, or only slightly increases, as the size of the mapped area increases. Debrecen, Hungary has the smallest FCM of 0.0357 for its 1 Km file. This count increases to 0.0400 for 2 Km, and then decreases to 0.0372 for 3 Km, a percentage increase of 12% and a percentage decrease of 7% respectively. Paris, France, perhaps unsurprisingly, has the highest FCM for a 1 Km file, at 1.8723. This FCM decreases to 1.8561 for 2 Km, and then increases to 1.8748 for 3 Km, a percentage decrease of 8% and a percentage increase of 1% respectively.

Our examination of these file attributes in relation to both file size and FCM indicates three important points:

1. File size generally increases as the grid size of a mapped area increases, irrespective of location.
2. The File Composition Measure of a file generally either decreases, or slightly increases with the grid size of a mapped area, irrespective of location.

The correlation between file size (in bytes) and FCM is not very strong. This is not particularly surprising when we consider how the FCM of a file is calculated. FCM is a functional relationship of Nodes and Ways in the file. File size is the total number of bytes in the file. If a file contains lots of tags – including tags which may not be relevant to OSM-CAT's analysis – these can significantly increase the size of a file. Also, Relations, while we do not consider them when calculating our FCM, could contain references to hundreds of Ways. Thus, expecting a perfectly monotonically increasing correlation relationship between FCM and file size is probably not realistic. Regardless, this does not affect our analysis.

It is important that the points mentioned above are considered when assessing the performance of OSM-CAT. In our opinion, OSM-CAT's efficiency is best assessed by measuring the time taken to process an OSM-XML file against our two key metrics: file size and File Composition Measure.

5.2.1 File Sizes & Processing Times

OSM-CAT performed much as expected with regards to file size and processing time, in that the time taken to process the data in a file increased as the file size increased. **Figure 5.2**, in which the file size of each of our 174 OSM-XML files is plotted against its corresponding processing time, shows this relationship. We see that, irrespective of the square grid size, the processing time generally increases as the file size increases. Also, we note that processing time has a strong relationship with file size, but is mostly unaffected by square grid size. For example, the 1 Km file of Stuttgart, Germany, with a file size of 3 MB, took 8.615 seconds to

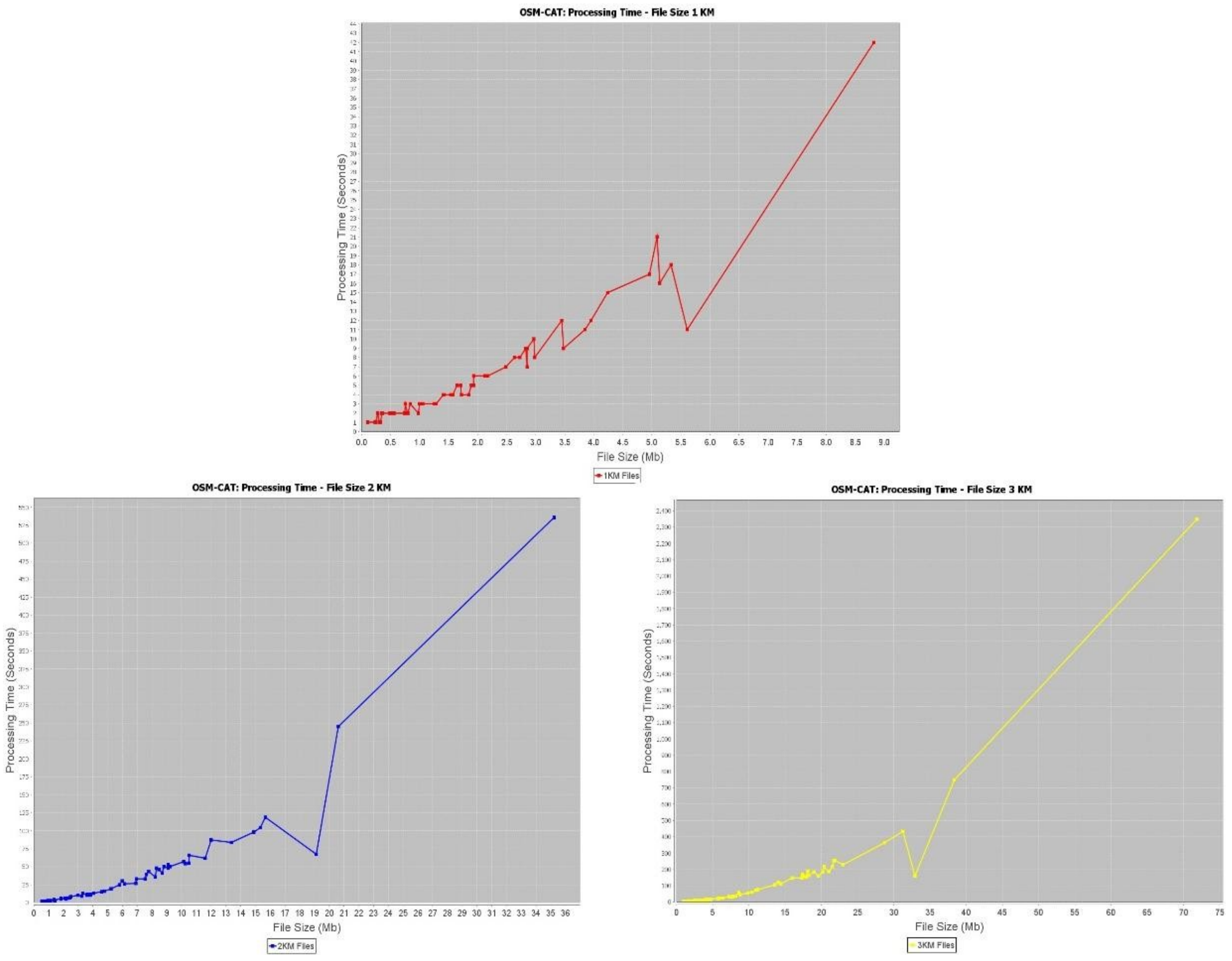


Figure 5.2: Relationship between OSM-CAT processing time and file size for 1 Km (red), 2 Km (blue), and 3 Km (yellow) square grid OSM-XML files

process. The 2 Km file of Edinburgh, Scotland, which also has a file size of 3 MB, took 10.057 seconds to process, just one second longer than the 1 Km Stuttgart file.

Figure 5.2 also highlights some interesting points with regard to processing time and file size. The large dip on the 1 Km, 2 Km and 3 Km lines is attributed to Copenhagen, Denmark, where file sizes of 5.6 MB, 19.1 MB, and 32.9 MB, did not result in the expected corresponding processing times, but rather in times of 11.974, 68.712, and 155.715 seconds respectively. Indeed, other locations with similar file sizes took notably longer to process. For example, the 1 Km file of Prague, Czech Republic, which has a smaller 1 Km size of 5 MB, took 17.417 seconds to process, over 5 seconds longer than the 1 Km Copenhagen file. This is interesting, and indicates a possible difference between data volunteered in Copenhagen, and in other locations. Indeed, compared to the Prague, Czech Republic OSM-XML file, we find that although Copenhagen's OSM-XML file is slightly larger, it contains far less OSM primitives in both its 2 Km and 3 Km files, but a far larger number of wikipedia linked places. Thus, Copenhagen's large file size might be due to data other than OSM primitive data which is not relevant to OSM-CAT's analysis, such as irrelevant tags. As OSM-CAT's analysis primarily involves the processing of OSM primitives, the Copenhagen exceptions may indicate a relationship between processing time, and number of OSM primitives in a file.

5.2.2 Complexity of OSM-XML Data (File Composition Measure)

Similar to OSM-CAT's performance in relation to file size, OSM-CAT performed much as expected with regards to processing time and a file's FCM. **Figure 5.3**, in which the FCM of each of our 174 OSM-XML files is plotted against its corresponding processing time, shows this relationship.

When the FCM of a file increases, so too does the processing time. However, unlike the relationship between processing time and file size, the relationship between processing time and FCM is slightly more complex. As shown in **Figure 5.3,** processing time is relative not only to the file's FCM, but also to the square grid size. For example, the 2 Km file of Bratislava, Slovakia, with a FCM of 0.7424, took 86.982 seconds to process. The 3 Km file of Karlsruhe, Germany, which has a similar FCM of 0.7454, took 267.21 seconds to process. Thus, two files with nearly equal FCMs, but of different square grid sizes, require vastly different processing times. This indicates that the relationship between processing time and FCM is not as strong

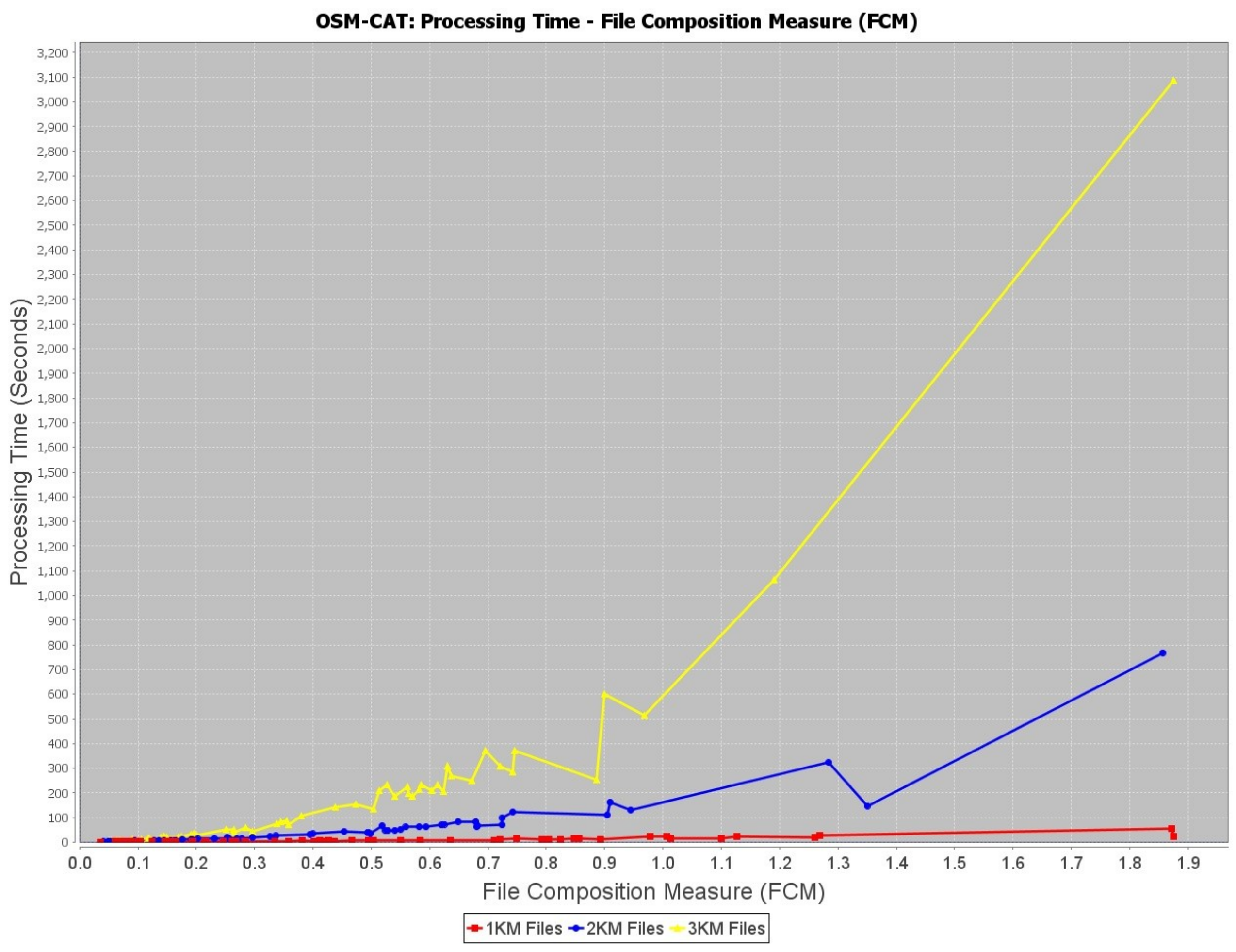


Figure 5.3: Relationship between Processing Time and File Composition Measure for 1 Km (red), 2 Km (blue), and 3 Km (yellow) square grid OSM-XML files

as that of processing time and file size. Indeed, examining the file sizes of Bratislava and Karlsruhe, we note a large difference, with sizes of 12 MB and 21.8 MB respectively.

Also, on the 2 Km line, we see that both Helsinki, Finland, with an FCM of 0.9445, and Salamanca, Spain, with an FCM of 1.3500, required similar processing times, with times of 97.225 and 100.076 seconds respectively. This is surprising, as two OSM-XML files with very different FCMs, are processed in times which differ only slightly. Examining the detailed reports of each, we see that the number of changesets in each file differs greatly, with counts of 2686 for Helsinki, and 632 for Salamanca. Thus, although the Salamanca file has a higher FCM than the Helsinki file, its low changeset count may be resulting in a processing time similar to that of Helsinki. This is plausible, as OSM-CAT's analysis involves the processing of changeset data. This may indicate a possible correlation between processing time and the changeset count of a file. Furthermore, the notable dip in the 3 Km line can also be attributed to Salamanca's low changeset count, again indicating a correlation between processing time and changeset count.

5.2.3 Validity and Completeness of Analysis Data

As per our project goals, the correctness and validity of the analysis data produced by OSM-CAT needs to be examined. A key feature of our application is the ability to analyse map feature tag validity. OSM-CAT examines each tag's key-value pair, and checks whether or not the pair is included in the *validtags.xml* file. If it is not, it is processed as invalid, and the invalid key-value pair is stored. OSM-CAT then generates an *invalidtagvalues.csv* file, outputting each of these key-value pairs. This allows users to examine all the invalid feature tag key-value pairs for an analysed OSM-XML file.

In our opinion, in order to successfully achieve our project goals, OSM-CAT should consistently produce correct analysis data. As a measure of whether or not OSM-CAT is doing so, we examine OSM-CAT's analysis of feature tag validity. OSM-CAT should only flag a tag key-value pair as invalid, if it is indeed invalid. Thus, for each of our 174 test corpus OSM-XML files, we checked each individual *invalidtagvalues.csv* file against the *validtags.xml* file. We found that OSM-CAT had correctly flagged and output to the *invalidtagvalues.csv* file only those key-value pairs which were not included in the *validtags.xml* file.

Moreover, we also examine the completeness of the output produced. OSM-CAT created a directory for each of the 174 OSM-XML files during our experimental analysis. Each directory was inspected, and we found that all expected files and charts were produced for each OSM-XML file. Each contained a directory of 13 chart JPEG images, a PDF report, an invalid tags report CSV file, and an error report LOG file. Next, the charts and reports were inspected for each of the 174 files. We found that all charts and reports were created successfully, and all contained valid analysis data. We considered analysis data valid if it corresponded to commonly expected output. Thus, we ensured no negative counts, no skewed charts, and no illegible data.

Overall, in our opinion, the results of our experimental analysis are promising. They reveal OSM-CAT's efficient processing of OSM-XML files, relative to both file size, and file complexity. No problems were encountered during our large scale experimental analysis, and OSM-CAT processed all files in the test corpus without producing any erroneous results, and without any unexpected exceptions. Any unexpected file contents, such as missing Contributor names or ids[14], were handled effectively, and all analysis reports produced were valid, and contained all elements specified by the user.

Additionally, to allow for a deeper understanding of the experimental analysis carried out in this section, and of OSM-CAT's analysis capabilities, we have made available the OSM-CAT analysis reports for a selection of European cities. Rather than unnecessarily lengthen the appendix section of the thesis, we have made these reports available for download. These PDF reports, along with the Invalid Tags CSV reports, are available on our web server [44].

In this section we presented our experimental analysis of OSM-CAT. In Section 5.1, we provided an overview of the experimental setup for this analysis, and we discussed the process of selecting our OSM-XML test corpus in Section 5.1.2. In Section 5.2, the results from our experimental analysis of OSM-CAT were detailed. With our proposed goals in mind, this analysis focused on both the efficiency of execution in relation to arbitrarily-sized, and arbitrarily-complex, files, and on the correctness and completeness of the produced analysis data. However, although we are very satisfied with OSM-CAT's performance overall, there is room for improvement, and indeed more work could be done to further develop OSM-CAT. We discuss our work, along with possible improvements to OSM-CAT, and potential for further development, in Section 6.

14 OpenStreetMap originally permitted anonymous edits, meaning edits with no form of attribution identifying a contributor. This was phased out completely in 2009.

6. Conclusion

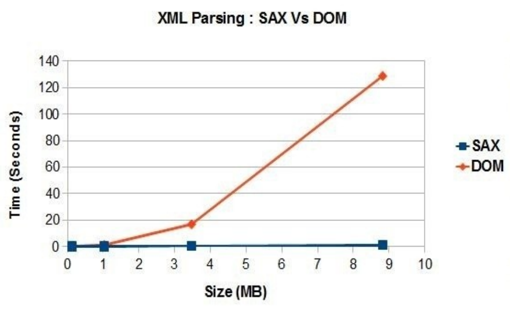
In this paper, we presented OSM-CAT, introducing a new, open-source application for performing contributor related analysis of OpenStreetMap XML data. We believe that through OSM-CAT's custom XML selective parsing technique, its intuitive graphical user interface, and its implementation of best practice, object-oriented, software development practices, the analysis of OSM-XML data is made considerably easier. Additionally, we recognise that the automated analysis methods of OSM-CAT can be further enhanced and extended. In Section 6.1, we present a critical analysis of our work, and in Section 6.2, we discuss possible future work.

6.1 Critical Analysis

In Sections 2.1 and 2.2, we investigated how XML data can be automatically processed using either of two Java parsing APIs – SAX and DOM. We discussed the differences between the two APIs, and we performed experimental analysis to decide which API was best suited to the task of processing arbitrarily-sized, and arbitrarily-complex OSM-XML data files. Our analysis indicated that SAX was the most efficient parsing API for this task, and in Section 3, we presented our software design, which adapted SAX, allowing us to build a custom, selective OSM-XML parser. This parser is the central processing module of OSM-CAT, and performs the bulk of the computations and analysis. In Section 4 the implementation of our custom parser was discussed.

Section 5 presented the experimental analysis of OSM-CAT. Keeping the focus on our first project goal – producing a tool which efficiently processes arbitrarily-sized, and arbitrarily-complex OSM-XML files, producing correct, valid analysis data. – we evaluated OSM-CAT by performing a full analysis on a test corpus of 174 OSM-XML files, and we measured the processing time against both the file size, and File Composition Measure (FCM), of each OSM-XML file. We also examined the correctness and completeness of the analysis data produced by OSM-CAT.

As discussed in Section 2.2, file size is a major issue when processing XML based data. In our opinion, the results of our experimental analysis confirm this. Our first proposed project goal specified an efficient data analysis application, and although, in our opinion, OSM-CAT



To the left, we see the results of basic processing for Debrecen, Dublin, Munich, and Paris, using both our test SAX parser, and our test DOM parser. This simple processing involved simply reading in and printing out the user names and Ids associated with the OSM primitives contained in the file. We chose to build the OSM-CAT OSM-XML parser using the SAX API.

As shown below in the 1, 2, and 3 KM result charts from our experimental analysis, the growth rate of OSM-CAT's processing time with respect to file size is closer to that of our DOM test parser.

However, it must be acknowledged the processing carried out by OSM-CAT is far more detailed, and involves a detailed analysis of almost all the spatial data contained in an OSM-XML file.

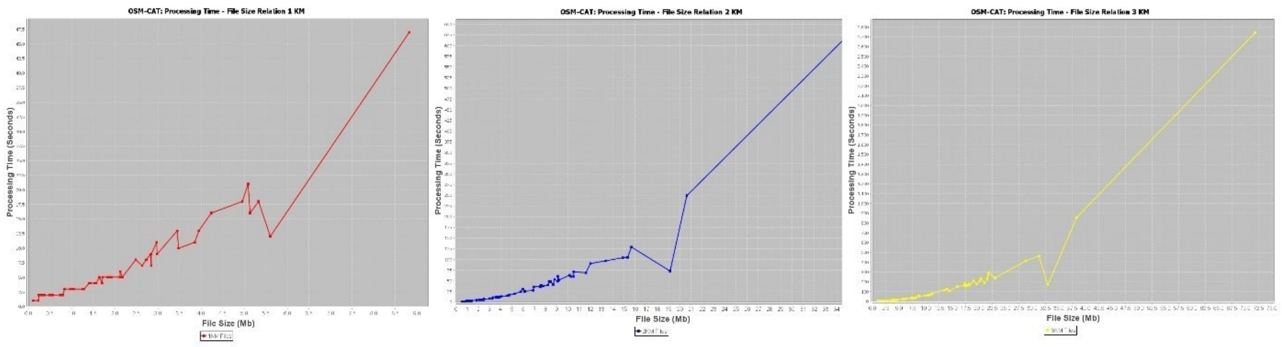


Figure 6.1: SAX & DOM basic processing compared to OSM-CAT full analysis processing

performs very well overall, its performance could be improved upon. To measure OSM-CAT's efficiency, we measured the processing time required for a full analysis of each of our 174 OSM-XML files. Now, using the analysis we performed on the SAX and DOM APIs in Section 2.2, we compare our initial goals with the final implementation. **Figure 6.1** shows both the initial test analysis using SAX and DOM, and the results of OSM-CAT's analysis, both in relation to file size.

For our implementation, we chose SAX , as our initial investigation showed it to be an efficient parsing API. Indeed, our initial experiments demonstrated a parsing time of OSM-XML files faster than that of DOM. As shown in **Figure 6.1**, our test SAX parser processing time demonstrated little growth with respect to file size. Therefore, we had intended that our implemented design would display similar growth, but as **Figure 6.1** shows, the growth rate of OSM-CAT's processing with respect to file size is greater than that of our test SAX parser.

However, it must be noted that our test SAX parser was processing only the contributor names and Ids associated with OSM primitives, whereas OSM-CAT is much more than a simple OSM-XML parser, as it performs a detailed analysis of almost all the spatial data contained in an OSM-XML file. Also, OSM-CAT is performing a lot of time-consuming and resource expensive I/O operations [45], creating numerous files of varying types. These factors need to be considered when analysing OSM-CAT's performance.

To further inspect OSM-CAT's performance, we need to consider the second major problem associated with processing OSM-XML data, that is, efficiently processing OSM-XML files of varying, arbitrary FCMs. As stated, we calculated a file's FCM based on the number of Nodes and Ways in the file. As the number of Nodes and Ways in a file increases, so too does the complexity. In Section 2.2.2, we calculated the complexity of the 1 Km OSM-XML files of Debrecen, Dublin, Munich, and Paris. Also, using our simple SAX parser from Section 2.2.1, the times required for basic processing of these files was measured. Then, in Section 5, the times required for a full analysis using OSM-CAT for each of the files was measured. Now, we compare the times required for basic processing with the times required for full, OSM-CAT analysis. **Table 6.1** shows this comparison.

File (1 Km)	Size	File Composition Measure (FCM)	Time – Basic (Seconds)	Time – Full (Seconds)
Debrecen, Hungary	115.8 KB	0.0357	0.113	1.680
Dublin, Ireland	1.0 MB	0.2672	0.320	3.057
Munich, Germany	3.5 MB	0.8931	0.540	10.366
Paris, France	8.8 MB	1.8723	1.206	43.186

Table 6.1: Comparison between processing times for basic and full analysis of selected 1 Km OSM-XML files

As shown, moving from basic processing of user names and Ids, to full processing of almost all the spatial data contained in a file, results in a longer processing time, irrespective of the file's FCM. Also, examining **Figure 5.3,** we note that although the processing time does increase with full OSM-CAT analysis, regardless of the FCM, the increases are greater for files of larger grid size. As larger grid sizes generally correspond to larger file sizes, this may indicate a stronger relationship between processing time and file size, than the relationship between processing time and FCM.

However, although full OSM-CAT analysis does require longer processing times than basic analysis, for files of any FCM, it must again be remembered that our test SAX parser performed extremely simply processing, compared to the detailed analysis carried out by OSM-CAT. In our opinion, OSM-CAT does efficiently process OSM-XML files of varying FCM, or complexity. Indeed, **Figure 5.3** shows that increases in processing times were gradual in relation to increases in FCM. Processing times which might be considered unreasonable were only encountered with the largest of our OSM-XML files, such as the 3 Km Paris, France file.

Additionally, in our analysis of OSM-CAT's performance, we examined the correctness and completeness of the analysis data produced. We found that OSM-CAT produced 100% correct analysis data across all 174 test corpus files, in relation to identifying invalid feature tag key-value pairs. This is exactly what we had intended, as, in our opinion, anything less than 100% correctness would be unacceptable. Also, we examined the output produced for all 174 OSM-XML files in terms of completeness, checking that each report and directory contained the analysis elements specified by the user. Again, we found that OSM-CAT had performed as intended, producing the complete, correct report set across each of the 174 files. These measures of correctness and completeness indicate the quality and consistency of OSM-CAT.

Furthermore, to truly draw conclusions on OSM-CAT, we also need to consider the second of our proposed goals – presenting users with a detailed, useful analysis of many aspects of OSM in the given mapped area. To examine the usefulness of our reports, we again look at the literature-based related work discussed in Section 2.3.1.

Girres and Touya [10] examined the quality of the French OSM dataset. They analysed the dataset using a selection of quality metrics, one of which is attribute accuracy. This involved the detailed analysis of tags. Such analysis can be performed quite easily using OSM-CAT, without the need for extensive knowledge of OSM features and tags. However, more detailed analysis regarding tags, such as examining the correlation between the number of contributors in an area, and the accuracy of tag values, could be provided by OSM-CAT, and will be noted as possible future work.

Flanagin and Metzger [16] examined VGI with regard to its "quality, reliability, and overall value". They analysed the credibility of VGI, and suggested that credibility is inherently difficult to measure with this method of information gathering. In our opinion, that credibility could be measured, in part, using OSM-CAT. Two of OSM-CAT's optional processing options, Tag Validity and Polygon Validity, directly pertain to credibility metrics, and provide some automated analysis of the quality and reliability of OSM data. Therefore, we believe that OSM-CAT's measuring of the quality and reliability of VGI is adequate and useful.

Mooney and Corcoran [7] investigated the annotating, or tagging, of spatial objects from the OSM databases. Overall, they concluded that “there are issues in how contributors tag and annotate spatial features in OSM”, and that these issues need to be addressed “before OSM can be considered for use in serious geomatics applications”. Again, we believe that OSM-CAT can contribute, as it provides detailed analysis of map features and their corresponding tag validity.

Moreover, Mooney, Corcoran et al. [11] investigated OSM data to develop measures of quality for OSM which operate in an unsupervised manner, and concluded that “the GIS community will require strong evidence of the quality of OSM data”, and that “OSM specific metrics are required”. In our opinion, the methods of experimental analysis presented in this paper could be made considerably easier with OSM-CAT's use. Particularly, the information generated by OSM-CAT regarding polygon and tag validity could be used to provide a more detailed investigation of OSM quality metrics.

M.Over et al. [8] reviewed the suitability and quality of available data. They concluded that the suitability of OSM data must be further investigated, “due to the pronounced regional differences in the quality of the dataset”. The quality of OSM data is again a central topic, and again, in our opinion, OSM-CAT's analysis directly pertains to this issue, and could greatly benefit such investigations.

Overall, we are very satisfied with OSM-CAT's performance. We acknowledge that there were a few files in our test corpus which caused OSM-CAT to require, what we have deemed to be, a very long time to process. However, for the most part OSM-CAT processed files in very acceptable times, some very quickly indeed. OSM-CAT produces correct, complete analysis data. It completely abstracts the details of OSM-XML from the user, and in our opinion, presents them with a detailed, useful analysis of many aspects of OSM data. Also, as per our proposal, OSM-CAT performs all of its analysis without the need for a database.

We also conclude that there is room for improvement. In particular, we acknowledge the need for analysis data of greater detail, and admit that our application could produce richer, more substantial reports, given the vast amount of data processed from an OSM-XML file. We discuss future work, and possible improvements, in Section 6.2.

6.2 Future Work

In this section, we discuss future work that we believe would improve upon the current version of OSM-CAT.

6.2.1 Performance

A key concern with the current version of OSM-CAT is its performance in relation to processing times. As discussed in Section 6.1, although OSM-CAT processed OSM-XML files quite efficiently, in our opinion, there is much room for improvement in this area. As software in general is often refactored and refined long after the completion of its first release, we believe that OSM-CAT could also greatly benefit from refactoring. Indeed, we would hope that future work would concentrate on improving OSM-CAT's performance, making it a more efficient application. Such work might involve improving OSM-CAT's processing time growth rate, bringing it more in line with that of our SAX test parser, discussed in 2.2.

6.2.2 Web-based & Mobile Computing

Currently OSM-CAT is desktop-based software. Future work will look to extend the functionality to a web-based version of OSM-CAT. This would also provide the possibility of providing a 'Web Map' interface for geographical area selection. Users might also be offered functionality to upload old versions of OSM-XML data where the changes in contributions over time can be compared with the current snapshot of the OSM data for that location. In the future we also intend to deploy OSM-CAT as a mobile Android[15] application, capable of downloading OSM-XML for the current location of the mobile device and performing analysis of that area in real-time.

6.2.3 More Detailed Analysis

A conclusion that we arrived at numerous times in Section 6.1 concerned the detail of the analysis produced by OSM-CAT. When examining the usefulness of our tool in relation to the related work discussed in Section 2.3.1, we found that although OSM-CAT provided a good level of detailed analysis, there is room for improvement. OSM-CAT could be further extended to produce analysis reports of greater detail. As we discussed, a major area of concern in relation to the OSM dataset is the validity and quality of its data. Studies into data validity and quality could greatly benefit from an application which provided extensive analysis reports.

15 Android is an operating system for mobile devices, such as smartphones.

6.2.4 New & Innovative Uses of Information

One possible future work extension might involve producing innovative visualisations or graphics, using OSM-CAT's analysis information. An example of such an innovative use of OSM data is shown in **Figure 6.2**. This is the work of an influential OSM mapper, Martijn van Exel [46]. He produces a 'Data Temperature' for OSM data. Basically, this involves colouring a map based on version number and date of last edit, per map feature. A map is then either 'hot' – very heavily edited recently, or 'cold' – not so recently edited, over all features. The data used to produce Martjin's map is easily calculated using OSM-CAT. Therefore producing similar map visualizations might be a possible future extension to OSM-CAT.

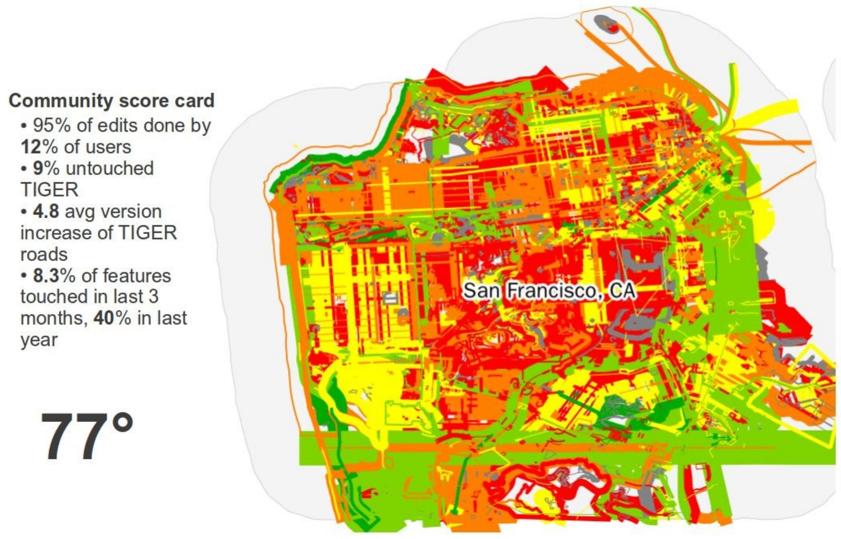

Community score card
• 95% of edits done by 12% of users
• 9% untouched TIGER
• 4.8 avg version increase of TIGER roads
• 8.3% of features touched in last 3 months, 40% in last year
77°
San Francisco, CA

Figure 6.2: Martijn van Exel's Data Temperature Visualisation

6.2.5 Improved User Interface

A key component of OSM-CAT is its GUI. It is essential that this GUI presents the user with a functional and intuitive interface. In our opinion, OSM-CAT provides such an interface. However, we also believe that more elements could be incorporated into the GUI. At the moment, OSM-CAT allows users to specify a mapped area to analyse by either selecting the OSM-XML file, or by entering latitude and longitude coordinates. One focus of future work might be to provide the user with a map interface. Using a map interface, users could navigate the Earth's geography, panning and zooming to specify their desired location for OSM-CAT analysis. Such an interface might look like **Figure 6.3**, which shows the Potlatch 2 [40] application.

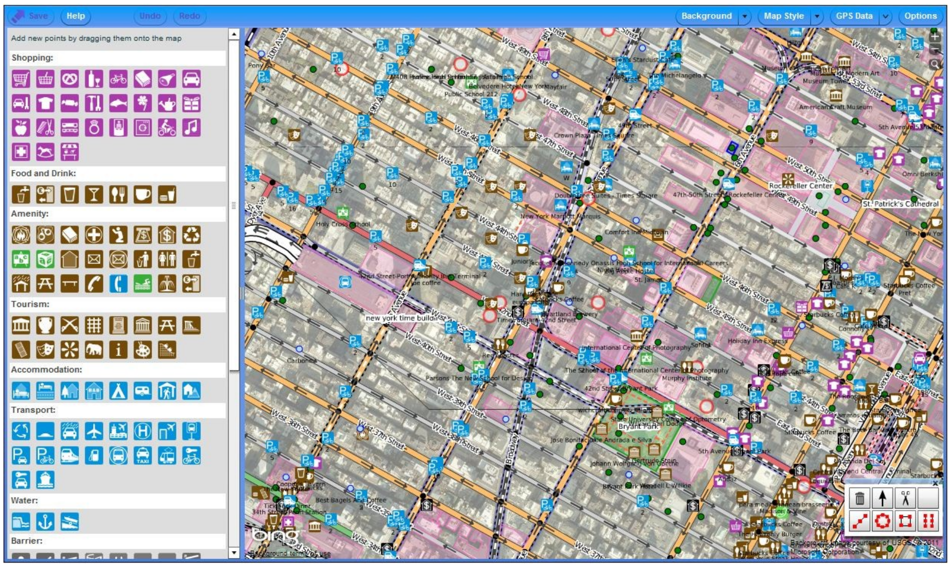


Figure 6.3: Potlatch 2 map interface

In Section 6, we presented a critical analysis of OSM-CAT's performance. We concluded that, although we are more than satisfied with its performance, there is room for improvement. We identified specific areas, such as processing time, and the detail of analysis data produced, where improvement is required. In Section 6.2, we presented an overview of the work which both we, and hopefully others, will contribute to implementing in future versions of OSM-CAT. This work will seek to both address the issues discussed in Section 6.1, and to build on the current features of OSM-CAT.

Finally, to encourage future development, OSM-CAT, and its full source code, has been made available on the World-Wide Web as a free and open-source software project [44]. Also, we have prepared a short Eclipse Development Setup Guide, shown in **Appendix E**, to assist potential developers of OSM-CAT. We do so with the intention of ensuring OSM-CAT's continued improvement and development.

Bibliography

1:	Michael F. Goodchild (2007). Citizens as Sensors: the world of volunteered geography. GeoJournal, Volume 69, pp. 211-221

2:	D Tapscott, D A Williams (2007). Wikinomics: How Mass Collaboration Changes Everything. Portfolio Hardcover, ISBN:1591841380

3:	OpenStreetMap Homepage (Last accessed 29th January 2012). http://www.openstreetmap.org/

4:	Extensible Markup Language (XML) (Last accessed 29th January 2012). http://www.w3.org/XML/

5:	OpenStreetMap XML (Last accessed 29th January 2012). http://wiki.openstreetmap.org/wiki/OSM_XML

6:	Nicole Ostlaender, Robin S. Smith, Bertrand De Longueville, Paul Smits (2010). What Volunteered Geographic Information is (good for) - designing a methodology for comparative analysis of existing applications to classify VGI and its uses. Geoscience and Remote Sensing Symposium (IGARSS). IEEE International, pp. 1422-1425

7:	Peter Mooney, Padraig Corcoran (2012). Annotating Spatial Features in OpenStreetMap. To Appear - Transactions in GIS, April 2012

8:	M. Over, A. Schilling, S. Neubauer, A. Zipf (2010). Generating web-based 3D City Models from OpenStreetMap: The Current situation in Germany. Computers, Environment and Urban Systems, Volume 34, pp. 496-507

9:	Paris, France in OpenStreetMap (Last accessed 29th January 2012). http://www.openstreetmap.org/?lat=48.85339&lon=2.3488&zoom=15&layers=M

10:	Jean-Francois Girres, Guillaume Touya (2010). Quality Assessment of the French OpenStreetMap Dataset. Transactions in GIS, Volume 14, pp. 435-459

11:	Peter Mooney, Padraig Corcoran, Adam C. Winstanley (2010). Towards Quality Metrics for OpenStreetMap. Proceedings of the 18th SIGSPATIAL International Conference on Advances in Geographic Information Systems, GIS '10, pp. 514-517

12:	OpenStreetMap Statistics (Last accessed 29th January 2012). http://wiki.openstreetmap.org/wiki/Stats

13:	Simple API for XML (SAX) Project (Last accessed 29th January 2012). http://www.saxproject.org/

14:	Document Object Model (DOM) (Last accessed 29th January 2012). http://www.w3.org/DOM/

15:	OpenStreetMap Primitives - Nodes, Ways, and Relations (Last accessed 29th January 2012). http://wiki.openstreetmap.org/wiki/Data_Primitives

16:	Andrew J. Flanagin, Miriam J. Metzger (2008). The Credibility of Volunteered Geographic

Information. GeoJournal, Volume 72, pp. 137-148

17: Martin Probst (2010). Processing Arbitrarily Large XML using a Persistent DOM. Proceedings of Balisage: The Markup Conference 2010. Balisage Series on Markup Technologies, Volume 5

18: James A Robinson (2009). Performance of XML-based applications: a case-study. Proceedings of the International Symposium on Processing XML Efficiently: Overcoming Limits on Space, Time, or Bandwidth. Balisage Series on Markup Technologies, Volume 4

19: Debrecen, Hungary in Wikipedia: The Free Encyclopedia (Last accessed 29th January 2012). http://en.wikipedia.org/wiki/Debrecen#Population

20: Dublin, Ireland in Wikipedia: The Free Encyclopedia (Last accessed 29th January 2012). http://en.wikipedia.org/wiki/Dublin

21: Munich, Germany in Wikipedia: The Free Encyclopedia (Last accessed 29th January 2012). http://en.wikipedia.org/wiki/Munich

22: Paris, France in Wikipedia: The Free Encyclopedia (Last accessed 29th January 2012). http://en.wikipedia.org/wiki/Paris

23: OpenStreetMap Database Statistics (Last accessed 29th January 2012). http://www.openstreetmap.org/stats/data_stats.html

24: PostgreSQL database (Last accessed 29th January 2012). http://wiki.openstreetmap.org/wiki/Osm2pgsql/schema

25: Normalisation in Wikipedia: The Free Encyclopedia (Last accessed 29th January 2012). http://en.wikipedia.org/wiki/Normalization_%28statistics%29

26: OpenStreetMap Projects (Last accessed 29th January 2012). http://wiki.openstreetmap.org/wiki/Category:Projects

27: AltogetherLost Contributors, Nodes, and Ways : Statistics (Last accessed 29th January 2012). http://osmstats.altogetherlost.com/

28: OSM Fight - Compare OSM contributors (Last accessed 29th January 2012). http://osmfight.neis-one.org/

29: OSM-HDYC - How did you Contribute to OSM? (Last accessed 29th January 2012). http://hdyc.neis-one.org/

30: Osmosis (Last accessed 29th January 2012). http://wiki.openstreetmap.org/wiki/Osmosis

31: JFreeChart (Last accessed 29th January 2012). http://www.jfree.org/jfreechart/

32: iText (Last accessed 29th January 2012). http://itextpdf.com/

33: Java HTML Generator (Last accessed 29th January 2012). http://artho.com/webtools/java/

34: Java OpenStreetMap Editor (Last accessed 29th January 2012). http://josm.openstreetmap.de/

35: Java Topology Suite (Last accessed 29th January 2012). http://www.vividsolutions.com/jts/JTSHome.htm

36: OpenStreetMap Changesets (Last accessed 29th January 2012).

http://wiki.openstreetmap.org/wiki/Changeset

37: OpenStreetMap Map Features Ontology (Last accessed 29th January 2012). http://wiki.openstreetmap.org/wiki/Map_Features

38: OpenStreetMap Taginfo (Last accessed 29th January 2012). http://taginfo.openstreetmap.org/

39: OpenStreetMap Planet.osm (Last accessed 29th January 2012). http://wiki.openstreetmap.org/wiki/Planet.osm

40: Potlatch 2 (Last accessed 29th January 2012). http://wiki.openstreetmap.org/wiki/Potlatch_2

41: Maynooth, Ireland in Wikipedia: The Free Encyclopedia (Last accessed 29th January 2012). http://en.wikipedia.org/wiki/Maynooth

42: European Cities in Wikipedia: The Free Encyclopedia (Last accessed 29th January 2012). http://en.wikipedia.org/wiki/List_of_cities_in_Europe

43: OSM History Splitter (Last accessed 29th January 2012). https://github.com/MaZderMind/osm-history-splitter

44: OSM-CAT Reports, Source Code, and Required Libraries (Last accessed 29th January 2012). http://www.cs.nuim.ie/~pmooney/OSM-CAT

45: Tuning Java I/O Performance (Last accessed 29th January 2012). http://java.sun.com/developer/technicalArticles/Programming/PerfTuning/

46: Taking the Temperature of local OpenStreetMap Communities (Last accessed 29th January 2012). http://oegeo.wordpress.com/2011/09/19/taking-the-temperature-of-local-openstreetmap-communities/

47: Eclipse Integrated Development Environment (IDE) (Last accessed 29th January 2012). http://www.eclipse.org/

48: Apache Maven (Last accessed 29th January 2012). http://maven.apache.org/

Appendix A: UML Class Diagrams

A.1: OSM-CAT UML Class Diagram

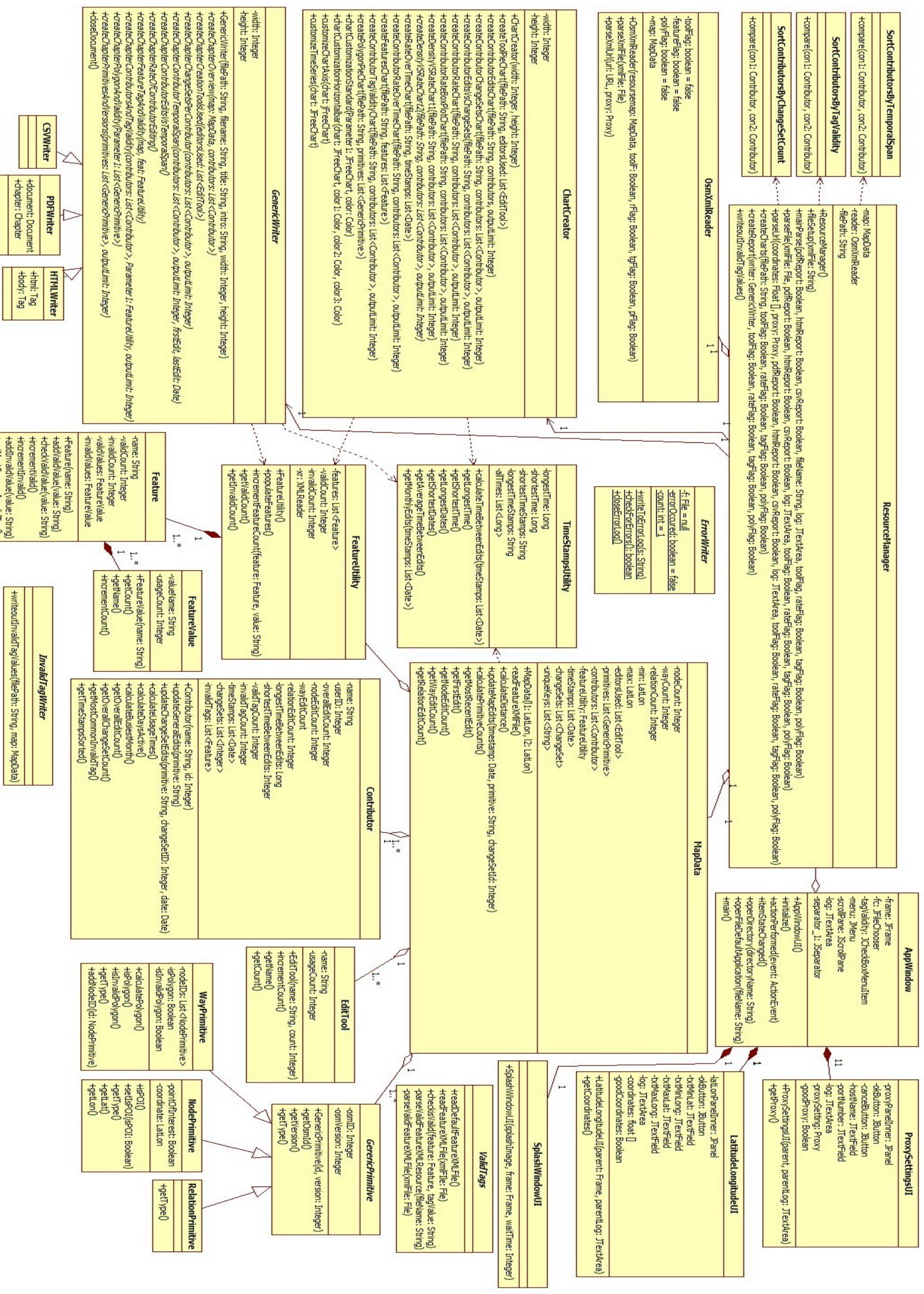


Diagram A.1: Complete OSM-CAT UML Class Diagram

A.2: *I/O* Module UML Class Diagram

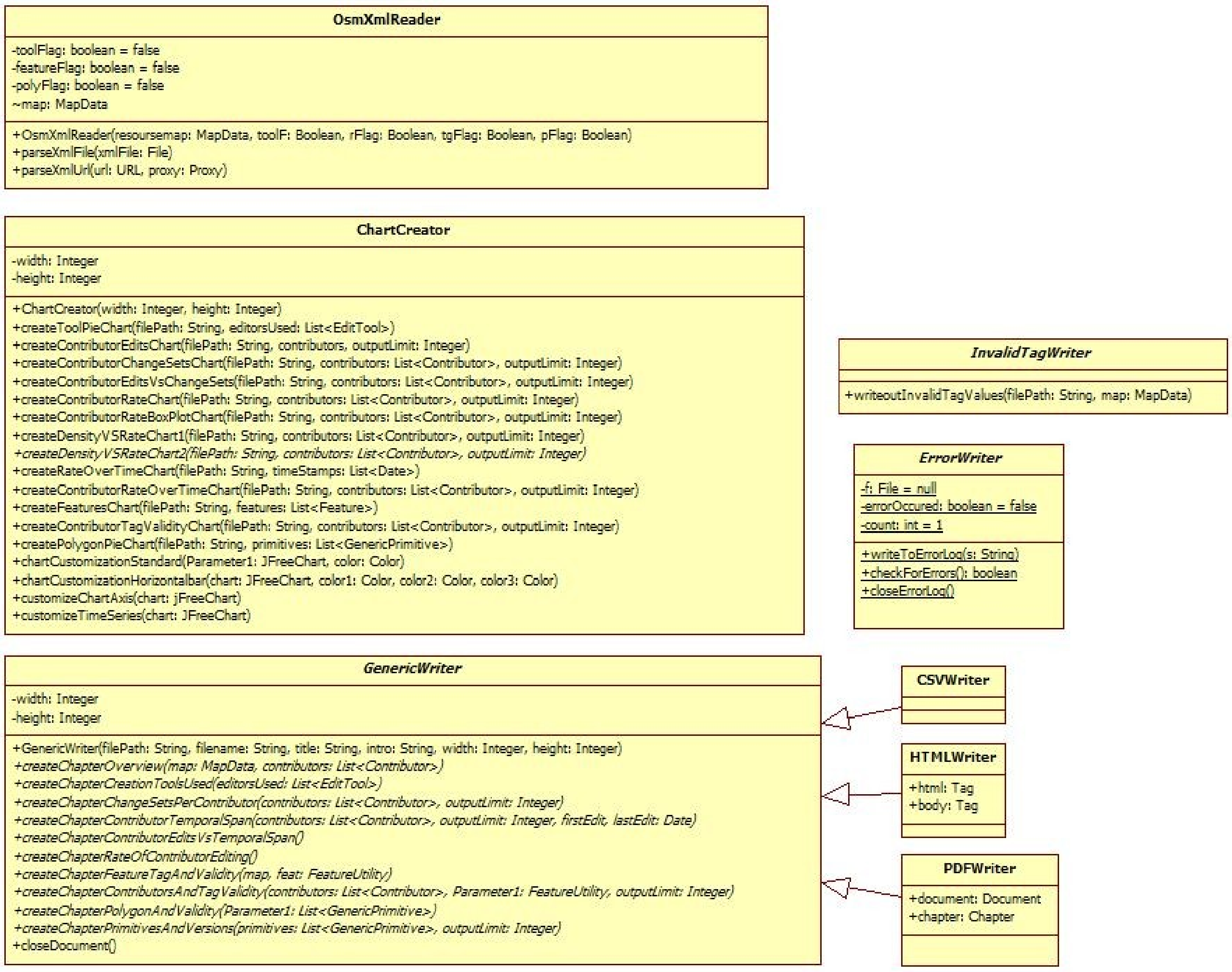

OsmXmlReader
-toolFlag: boolean = false
-featureFlag: boolean = false
-polyFlag: boolean = false
~map: MapData
+OsmXmlReader(resoursemap: MapData, toolF: Boolean, rFlag: Boolean, tgFlag: Boolean, pFlag: Boolean)
+parseXmlFile(xmlFile: File)
+parseXmlUrl(url: URL, proxy: Proxy)
ChartCreator
-width: Integer
-height: Integer
+ChartCreator(width: Integer, height: Integer)
+createToolPieChart(filePath: String, editorsUsed: List<EditTool>)
+createContributorEditsChart(filePath: String, contributors, outputLimit: Integer)
+createContributorChangeSetsChart(filePath: String, contributors: List<Contributor>, outputLimit: Integer)
+createContributorEditsVsChangeSets(filePath: String, contributors: List<Contributor>, outputLimit: Integer)
+createContributorRateChart(filePath: String, contributors: List<Contributor>, outputLimit: Integer)
+createContributorRateBoxPlotChart(filePath: String, contributors: List<Contributor>, outputLimit: Integer)
+createDensityVSRateChart1(filePath: String, contributors: List<Contributor>, outputLimit: Integer)
+createDensityVSRateChart2(filePath: String, contributors: List<Contributor>, outputLimit: Integer)
+createRateOverTimeChart(filePath: String, timeStamps: List<Date>)
+createContributorRateOverTimeChart(filePath: String, contributors: List<Contributor>, outputLimit: Integer)
+createFeaturesChart(filePath: String, features: List<Feature>)
+createContributorTagValidityChart(filePath: String, contributors: List<Contributor>, outputLimit: Integer)
+createPolygonPieChart(filePath: String, primitives: List<GenericPrimitive>)
+chartCustomizationStandard(Parameter1: JFreeChart, color: Color)
+chartCustomizationHorizontalbar(chart: JFreeChart, color1: Color, color2: Color, color3: Color)
+customizeChartAxis(chart: jFreeChart)
+customizeTimeSeries(chart: JFreeChart)
InvalidTagWriter
+writeoutInvalidTagValues(filePath: String, map: MapData)
ErrorWriter
-f: File = null
-errorOccured: boolean = false
-count: int = 1
+writeToErrorLog(s: String)
+checkForErrors(): boolean
+closeErrorLog()
GenericWriter
-width: Integer
-height: Integer
+GenericWriter(filePath: String, filename: String, title: String, intro: String, width: Integer, height: Integer)
+createChapterOverview(map: MapData, contributors: List<Contributor>)
+createChapterCreationToolsUsed(editorsUsed: List<EditTool>)
+createChapterChangeSetsPerContributor(contributors: List<Contributor>, outputLimit: Integer)
+createChapterContributorTemporalSpan(contributors: List<Contributor>, outputLimit: Integer, firstEdit, lastEdit: Date)
+createChapterContributorEditsVsTemporalSpan()
+createChapterRateOfContributorEditing()
+createChapterFeatureTagAndValidity(map, feat: FeatureUtility)
+createChapterContributorsAndTagValidity(contributors: List<Contributor>, Parameter1: FeatureUtility, outputLimit: Integer)
+createChapterPolygonAndValidity(Parameter1: List<GenericPrimitive>)
+createChapterPrimitivesAndVersions(primitives: List<GenericPrimitive>, outputLimit: Integer)
+closeDocument()
CSVWriter
HTMLWriter
+html: Tag
+body: Tag
PDFWriter
+document: Document
+chapter: Chapter

Diagram A.2: Complete *I/O* module UML Class Diagram

A.3: *Data* and *MapElements* Modules UML Class Diagram

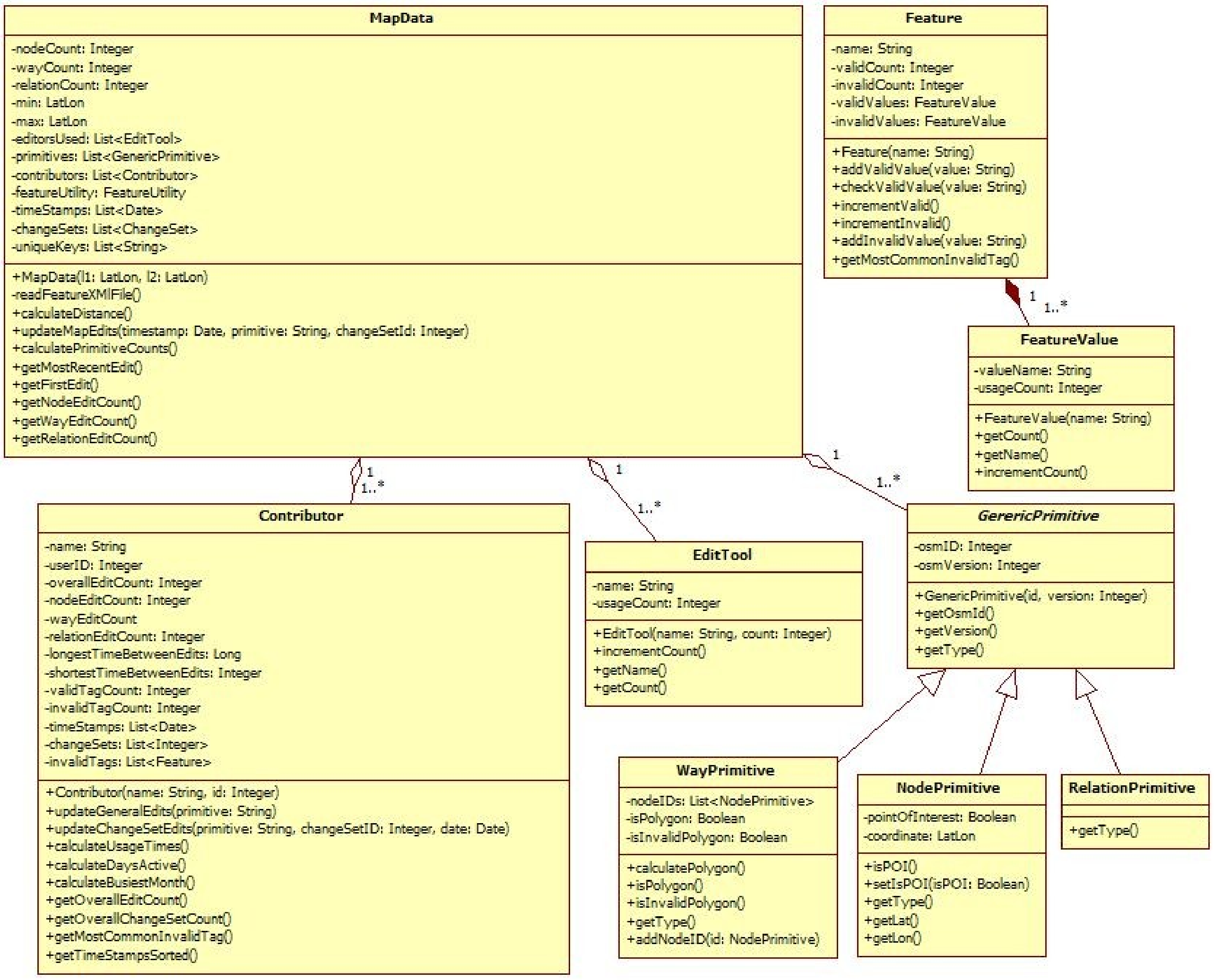

MapData
-nodeCount: Integer
-wayCount: Integer
-relationCount: Integer
-min: LatLon
-max: LatLon
-editorsUsed: List<EditTool>
-primitives: List<GenericPrimitive>
-contributors: List<Contributor>
-featureUtility: FeatureUtility
-timeStamps: List<Date>
-changeSets: List<ChangeSet>
-uniqueKeys: List<String>
+MapData(l1: LatLon, l2: LatLon)
-readFeatureXMlFile()
+calculateDistance()
+updateMapEdits(timestamp: Date, primitive: String, changeSetId: Integer)
+calculatePrimitiveCounts()
+getMostRecentEdit()
+getFirstEdit()
+getNodeEditCount()
+getWayEditCount()
+getRelationEditCount()
Feature
-name: String
-validCount: Integer
-invalidCount: Integer
-validValues: FeatureValue
-invalidValues: FeatureValue
+Feature(name: String)
+addValidValue(value: String)
+checkValidValue(value: String)
+incrementValid()
+incrementInvalid()
+addInvalidValue(value: String)
+getMostCommonInvalidTag()
1
1..*
FeatureValue
-valueName: String
-usageCount: Integer
+FeatureValue(name: String)
+getCount()
+getName()
+incrementCount()
1
1..*
1
1..*
1
1..*
Contributor
-name: String
-userID: Integer
-overallEditCount: Integer
-nodeEditCount: Integer
-wayEditCount
-relationEditCount: Integer
-longestTimeBetweenEdits: Long
-shortestTimeBetweenEdits: Integer
-validTagCount: Integer
-invalidTagCount: Integer
-timeStamps: List<Date>
-changeSets: List<Integer>
-invalidTags: List<Feature>
+Contributor(name: String, id: Integer)
+updateGeneralEdits(primitive: String)
+updateChangeSetEdits(primitive: String, changeSetID: Integer, date: Date)
+calculateUsageTimes()
+calculateDaysActive()
+calculateBusiestMonth()
+getOverallEditCount()
+getOverallChangeSetCount()
+getMostCommonInvalidTag()
+getTimeStampsSorted()
EditTool
-name: String
-usageCount: Integer
+EditTool(name: String, count: Integer)
+incrementCount()
+getName()
+getCount()
GerericPrimitive
-osmID: Integer
-osmVersion: Integer
+GenericPrimitive(id, version: Integer)
+getOsmId()
+getVersion()
+getType()
WayPrimitive
-nodeIDs: List<NodePrimitive>
-isPolygon: Boolean
-isInvalidPolygon: Boolean
+calculatePolygon()
+isPolygon()
+isInvalidPolygon()
+getType()
+addNodeID(id: NodePrimitive)
NodePrimitive
-pointOfInterest: Boolean
-coordinate: LatLon
+isPOI()
+setIsPOI(isPOI: Boolean)
+getType()
+getLat()
+getLon()
RelationPrimitive
+getType()

Diagram A.3: Complete *Data* and *MapElements* modules UML Class Diagram

A.4: *GUI* Module UML Class Diagram

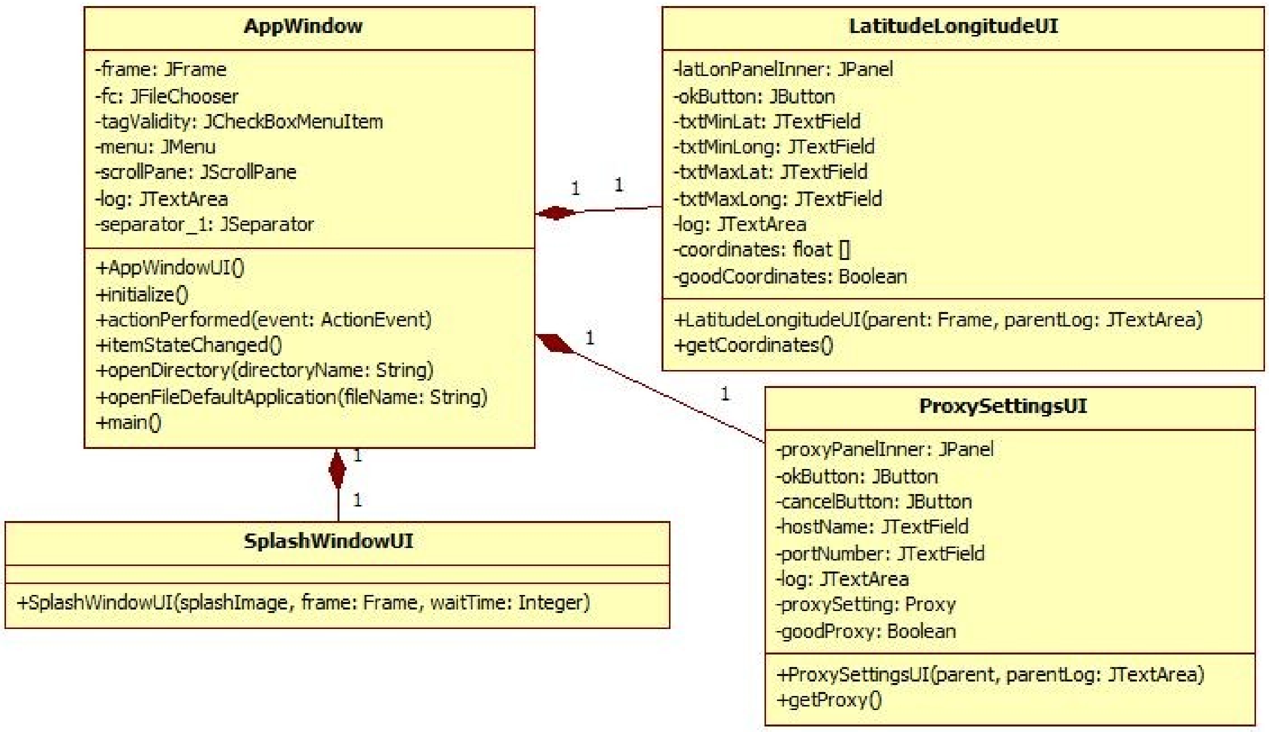


Diagram A.4: Complete *GUI* module UML Class Diagram

A.5: *Utilities* Module UML Class Diagram

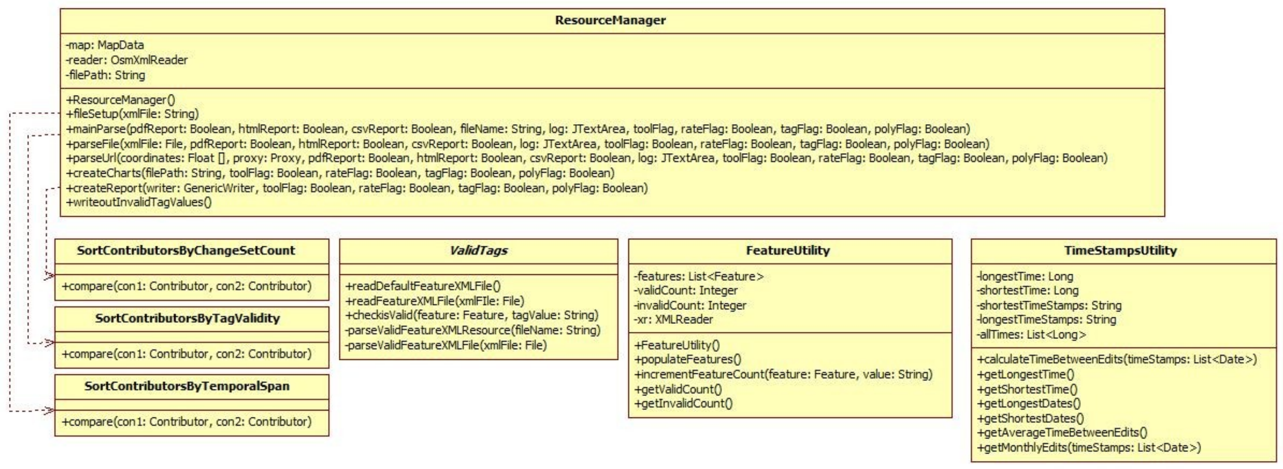
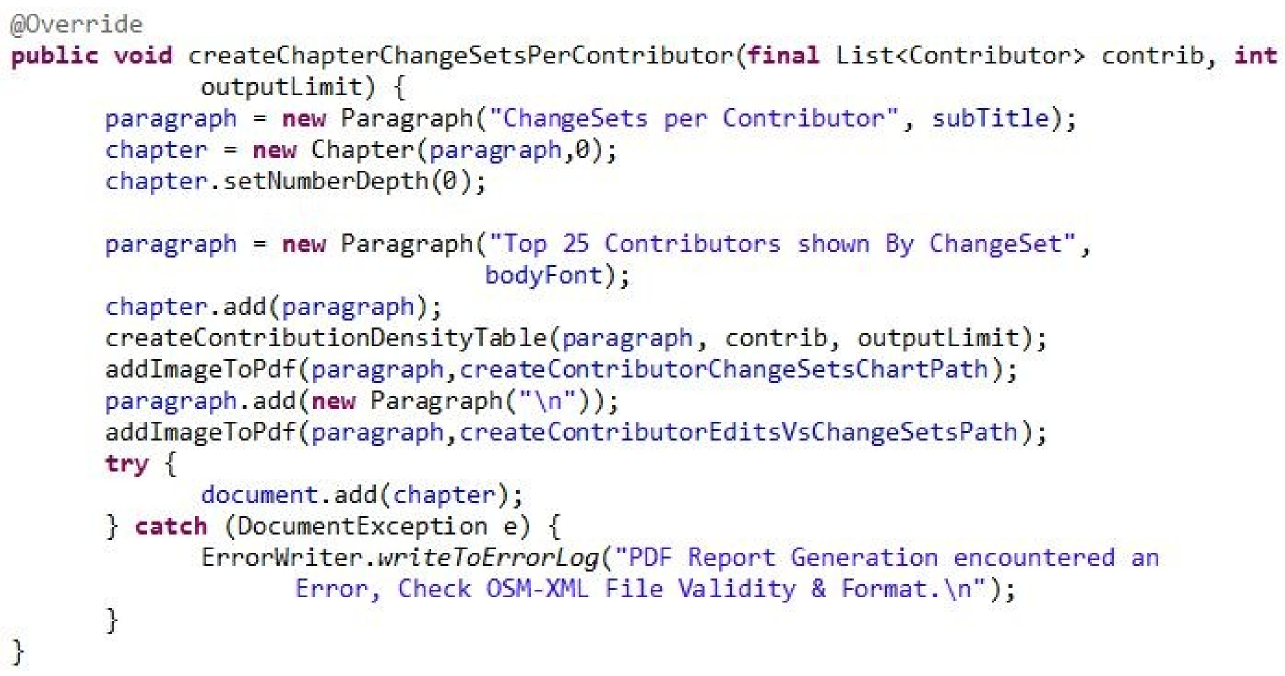

ResourceManager
-map: MapData
-reader: OsmXmlReader
-filePath: String
+ResourceManager()
+fileSetup(xmlFile: String)
+mainParse(pdfReport: Boolean, htmlReport: Boolean, csvReport: Boolean, fileName: String, log: JTextArea, toolFlag, rateFlag: Boolean, tagFlag: Boolean, polyFlag: Boolean)
+parseFile(xmlFile: File, pdfReport: Boolean, htmlReport: Boolean, csvReport: Boolean, log: JTextArea, toolFlag: Boolean, rateFlag: Boolean, tagFlag: Boolean, polyFlag: Boolean)
+parseUrl(coordinates: Float [], proxy: Proxy, pdfReport: Boolean, htmlReport: Boolean, csvReport: Boolean, log: JTextArea, toolFlag: Boolean, rateFlag: Boolean, tagFlag: Boolean, polyFlag: Boolean)
+createCharts(filePath: String, toolFlag: Boolean, rateFlag: Boolean, tagFlag: Boolean, polyFlag: Boolean)
+createReport(writer: GenericWriter, toolFlag: Boolean, rateFlag: Boolean, tagFlag: Boolean, polyFlag: Boolean)
+writeoutInvalidTagValues()
SortContributorsByChangeSetCount
+compare(con1: Contributor, con2: Contributor)
SortContributorsByTagValidity
+compare(con1: Contributor, con2: Contributor)
SortContributorsByTemporalSpan
+compare(con1: Contributor, con2: Contributor)
ValidTags
+readDefaultFeatureXMLFile()
+readFeatureXMLFile(xmlFIle: File)
+checkisValid(feature: Feature, tagValue: String)
-parseValidFeatureXMLResource(fileName: String)
-parseValidFeatureXMLFile(xmlFile: File)
FeatureUtility
-features: List<Feature>
-validCount: Integer
-invalidCount: Integer
-xr: XMLReader
+FeatureUtility()
+populateFeatures()
+incrementFeatureCount(feature: Feature, value: String)
+getValidCount()
+getInvalidCount()
TimeStampsUtility
-longestTime: Long
-shortestTime: Long
-shortestTimeStamps: String
-longestTimeStamps: String
-allTimes: List<Long>
+calculateTimeBetweenEdits(timeStamps: List<Date>)
+getLongestTime()
+getShortestTime()
+getLongestDates()
+getShortestDates()
+getAverageTimeBetweenEdits()
+getMonthlyEdits(timeStamps: List<Date>)

Diagram A.5: Complete *Utilities* module UML Class Diagram

Appendix B: Java Code

B.1: iText PDF Chapter Creation

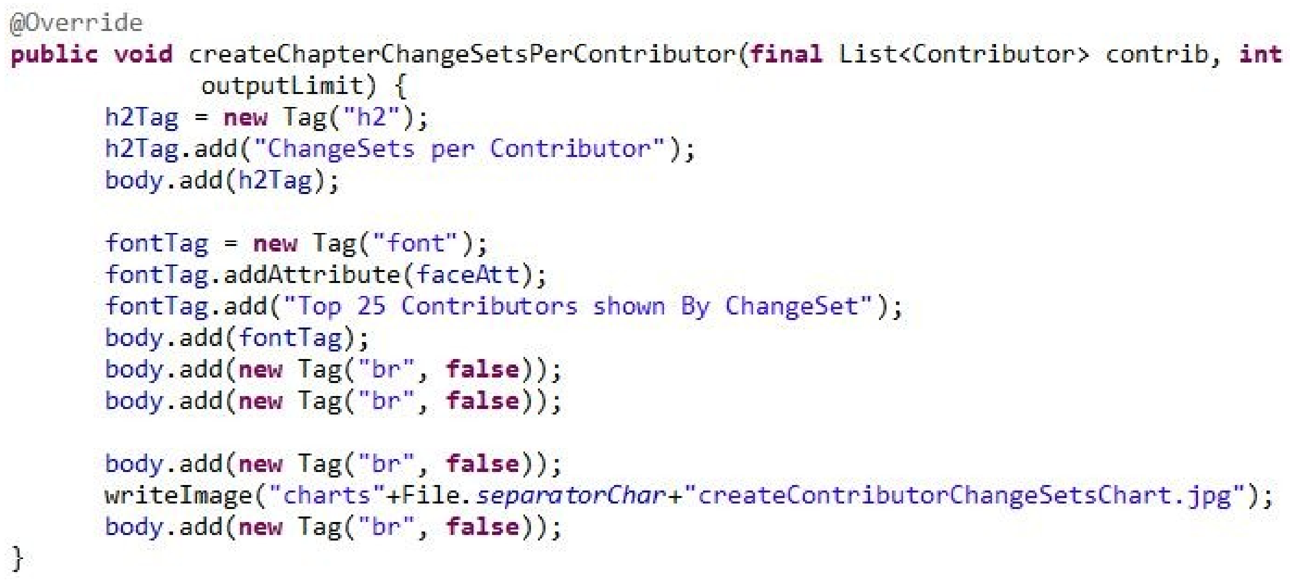


```java
@Override
public void createChapterChangeSetsPerContributor(final List<Contributor> contrib, int
            outputLimit) {
      paragraph = new Paragraph("ChangeSets per Contributor", subTitle);
      chapter = new Chapter(paragraph,0);
      chapter.setNumberDepth(0);

      paragraph = new Paragraph("Top 25 Contributors shown By ChangeSet",
                              bodyFont);
      chapter.add(paragraph);
      createContributionDensityTable(paragraph, contrib, outputLimit);
      addImageToPdf(paragraph,createContributorChangeSetsChartPath);
      paragraph.add(new Paragraph("\n"));
      addImageToPdf(paragraph,createContributorEditsVsChangeSetsPath);
      try {
            document.add(chapter);
      } catch (DocumentException e) {
            ErrorWriter.writeToErrorLog("PDF Report Generation encountered an
                  Error, Check OSM-XML File Validity & Format.\n");
      }
}
```

Code B.1: Creation of PDF chapter using iText

B.2: HTML Generator Section Creation



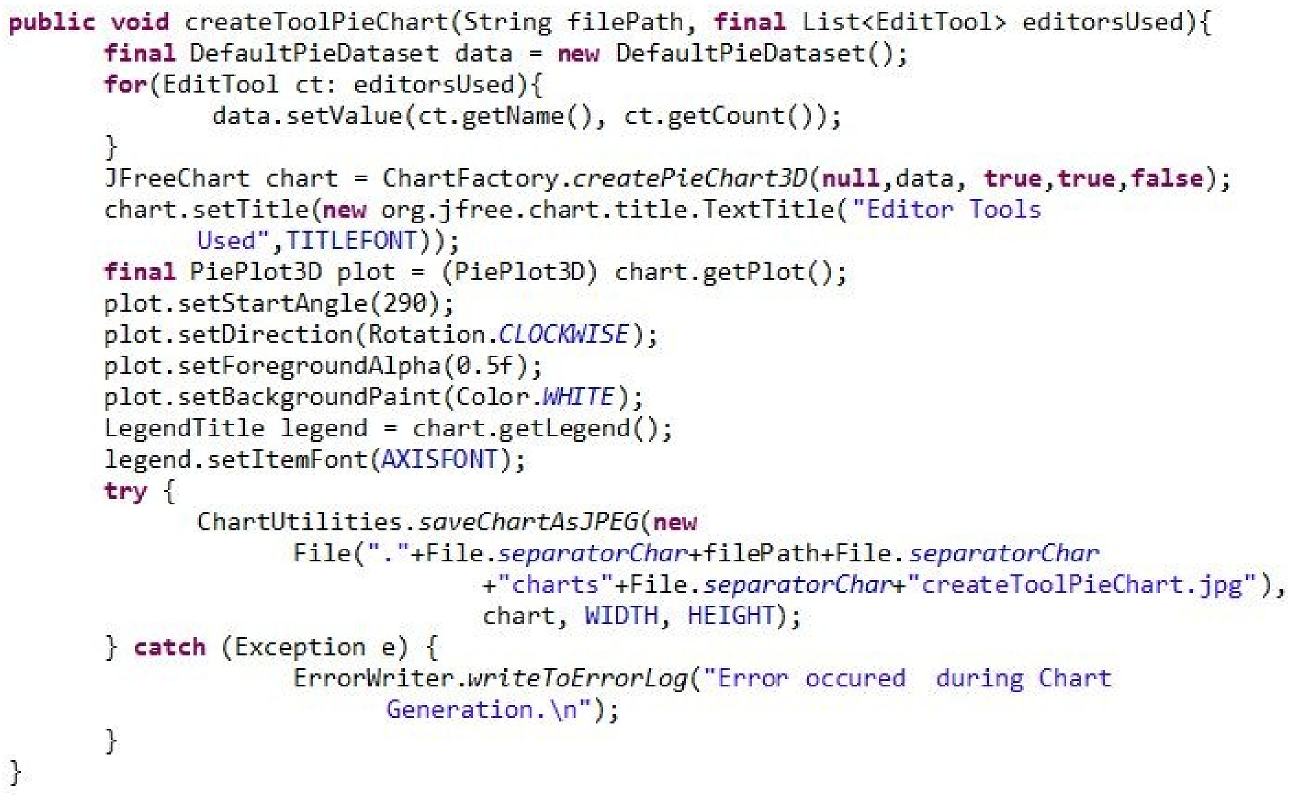
```java
@Override
public void createChapterChangeSetsPerContributor(final List<Contributor> contrib, int
            outputLimit) {
      h2Tag = new Tag("h2");
      h2Tag.add("ChangeSets per Contributor");
      body.add(h2Tag);

      fontTag = new Tag("font");
      fontTag.addAttribute(faceAtt);
      fontTag.add("Top 25 Contributors shown By ChangeSet");
      body.add(fontTag);
      body.add(new Tag("br", false));
      body.add(new Tag("br", false));

      body.add(new Tag("br", false));
      writeImage("charts"+File.separatorChar+"createContributorChangeSetsChart.jpg");
      body.add(new Tag("br", false));
}
```

Code B.2: Creation of HTML section using HTML Generator

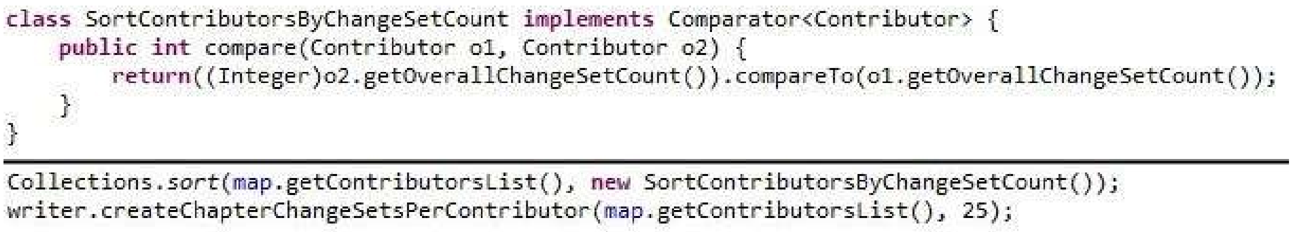
B.3: JFreeChart Chart Creation



```
public void createToolPieChart(String filePath, final List<EditTool> editorsUsed){
       final DefaultPieDataset data = new DefaultPieDataset();
       for(EditTool ct: editorsUsed){
              data.setValue(ct.getName(), ct.getCount());
       }
       JFreeChart chart = ChartFactory.createPieChart3D(null,data, true,true,false);
       chart.setTitle(new org.jfree.chart.title.TextTitle("Editor Tools
             Used",TITLEFONT));
       final PiePlot3D plot = (PiePlot3D) chart.getPlot();
       plot.setStartAngle(290);
       plot.setDirection(Rotation.CLOCKWISE);
       plot.setForegroundAlpha(0.5f);
       plot.setBackgroundPaint(Color.WHITE);
       LegendTitle legend = chart.getLegend();
       legend.setItemFont(AXISFONT);
       try {
             ChartUtilities.saveChartAsJPEG(new
                    File("."+File.separatorChar+filePath+File.separatorChar
                                 +"charts"+File.separatorChar+"createToolPieChart.jpg"),
                                 chart, WIDTH, HEIGHT);
       } catch (Exception e) {
                    ErrorWriter.writeToErrorLog("Error occured  during Chart
                          Generation.\n");
       }
}
```

Code B.3: Creation of chart using JFreeChart

B.4: Comparator Implementation and Use



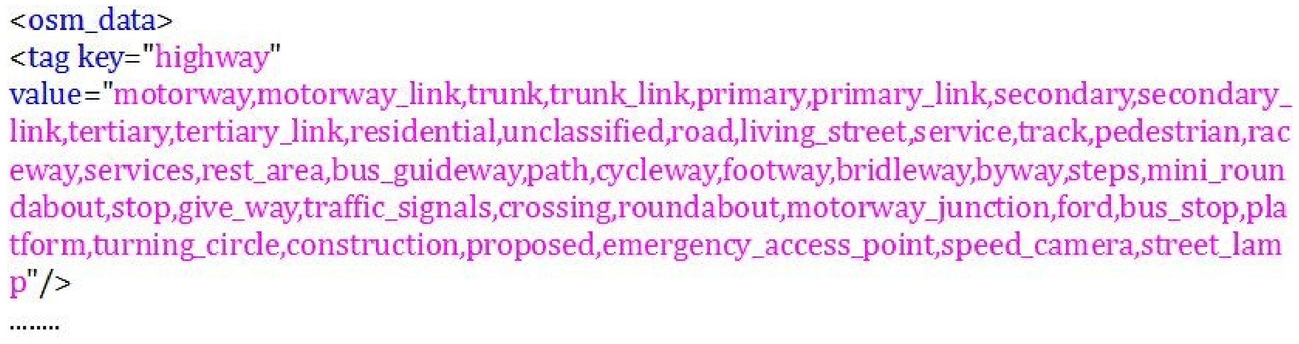
```
class SortContributorsByChangeSetCount implements Comparator<Contributor> {
    public int compare(Contributor o1, Contributor o2) {
        return((Integer)o2.getOverallChangeSetCount()).compareTo(o1.getOverallChangeSetCount());
    }
}
```

```
Collections.sort(map.getContributorsList(), new SortContributorsByChangeSetCount());
writer.createChapterChangeSetsPerContributor(map.getContributorsList(), 25);
```

Code B.4: *Comparator* implementation and use

Appendix C: Figures

C.1: Sub-section of Default XML Valid Tags File



```
<osm_data>
<tag key="highway"
value="motorway,motorway_link,trunk,trunk_link,primary,primary_link,secondary,secondary_
link,tertiary,tertiary_link,residential,unclassified,road,living_street,service,track,pedestrian,rac
eway,services,rest_area,bus_guideway,path,cycleway,footway,bridleway,byway,steps,mini_roun
dabout,stop,give_way,traffic_signals,crossing,roundabout,motorway_junction,ford,bus_stop,pla
tform,turning_circle,construction,proposed,emergency_access_point,speed_camera,street_lam
p"/>
........
```

Figure C.1: Sub-section of default XML valid tag file, *validtags.xml*

Appendix D: OSM-CAT Graphical User Interface Overview

The following is a complete overview of OSM-CAT's GUI. 'Greyed-out' buttons/text indicate disabled components.

Figure D.1: OSM-CAT logo, displayed when the application is launched

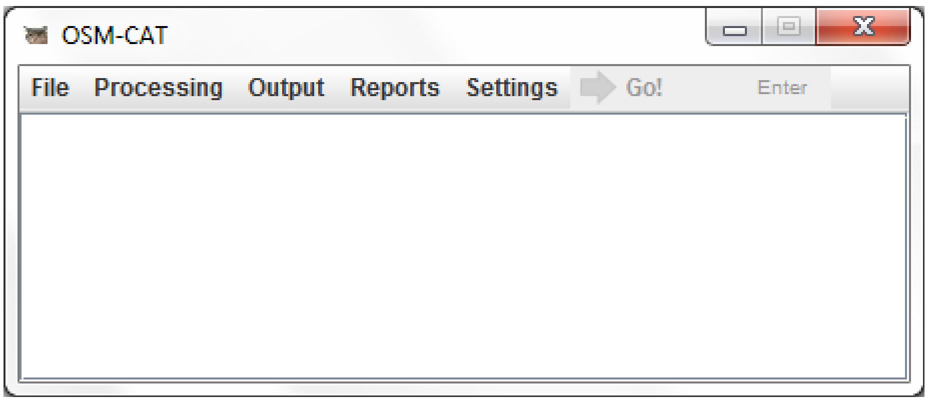


Figure D.2: OSM-CAT application window (No OSM-XML file selected)

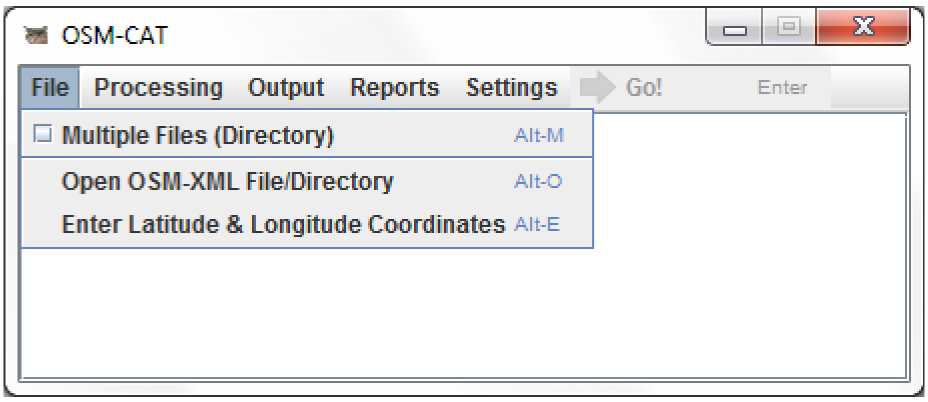

OSM-CAT
File
Processing
Output
Reports
Settings
Go!
Enter
Multiple Files (Directory) Alt-M
Open OSM-XML File/Directory Alt-O
Enter Latitude & Longitude Coordinates Alt-E

Figure D.3: OSM-CAT *File* menu

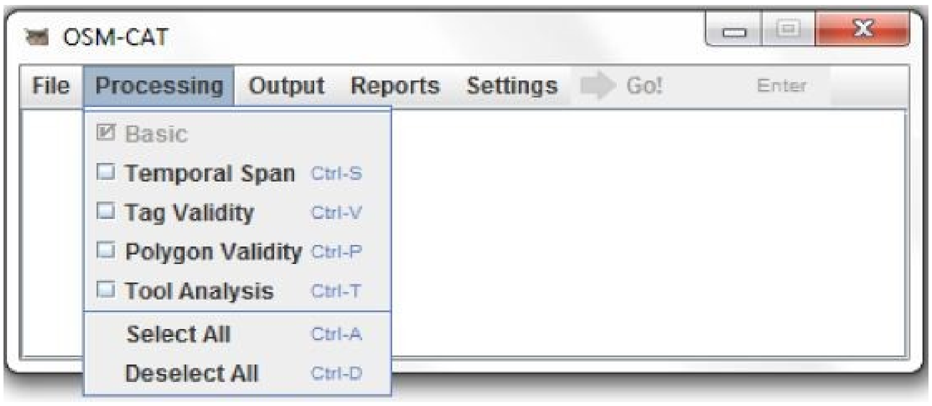


Figure D.4: OSM-CAT *Processing* menu

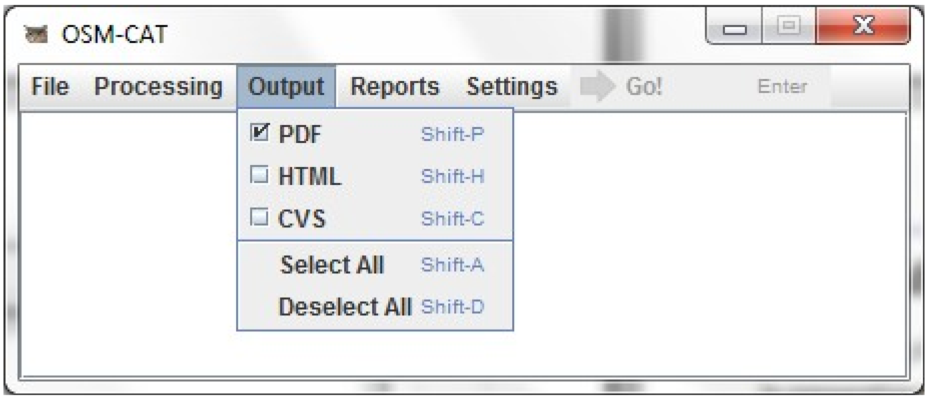


Figure D.5: OSM-CAT *Output* menu

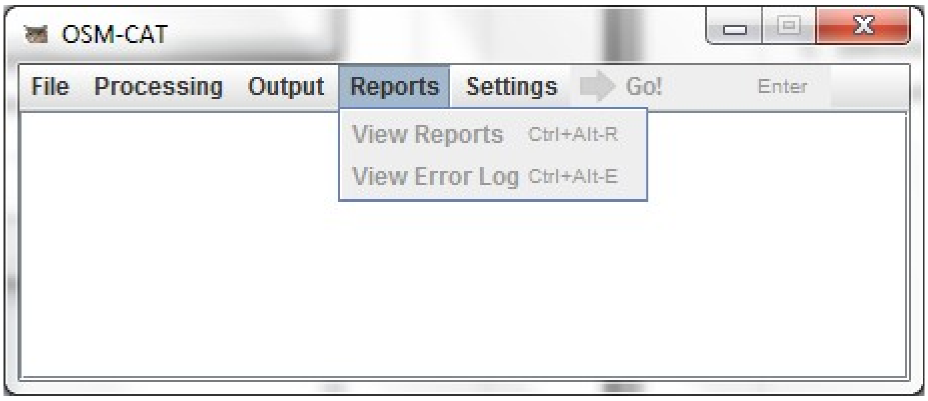


Figure D.6: OSM-CAT *Reports* menu

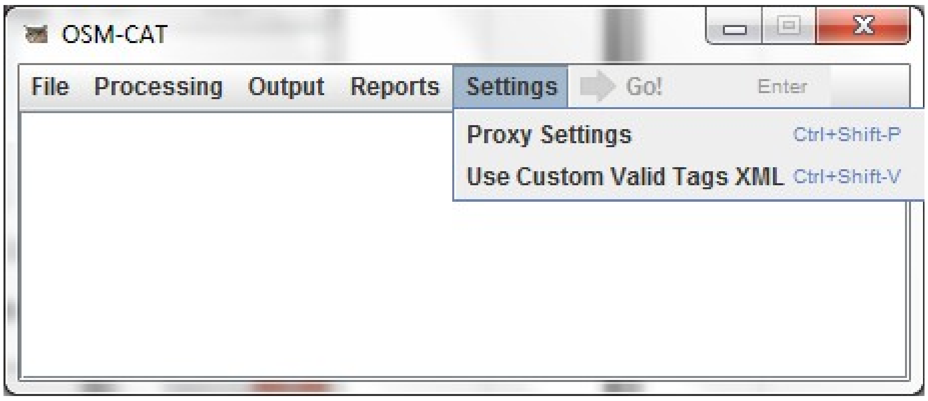
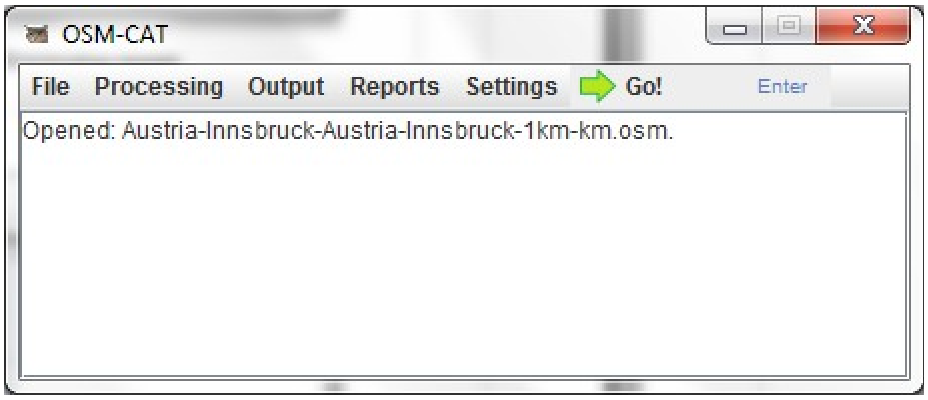
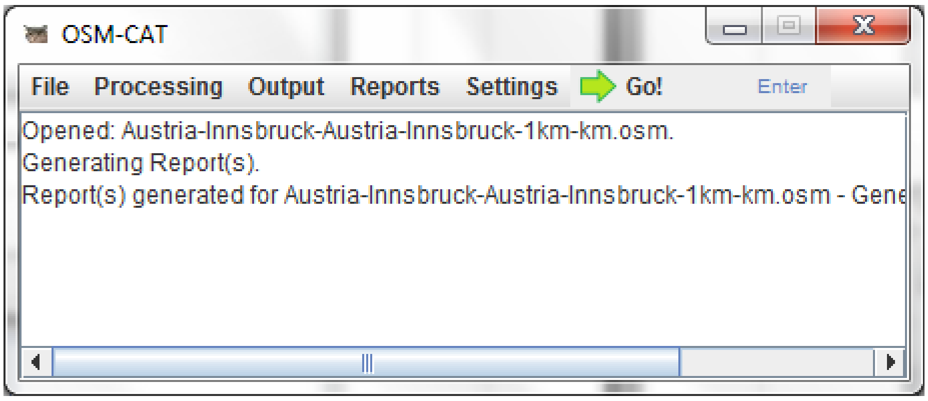

OSM-CAT
File
Processing
Output
Reports
Settings
Go!
Enter
Proxy Settings Ctrl+Shift-P
Use Custom Valid Tags XML Ctrl+Shift-V

Figure D.7: OSM-CAT *Settings* menu



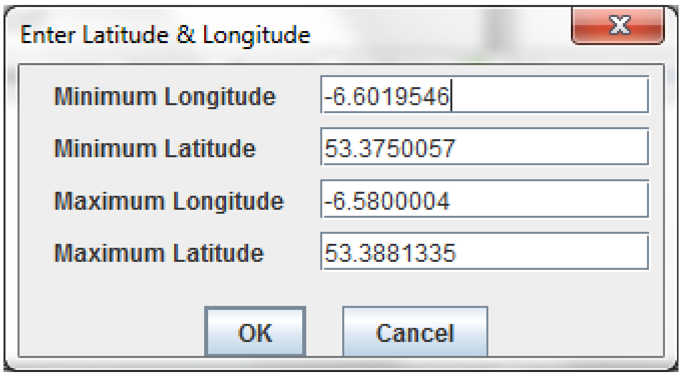
```
Opened: Austria-Innsbruck-Austria-Innsbruck-1km-km.osm.
```

Figure D.8: OSM-CAT application window (OSM-XML file selected)



```
Opened: Austria-Innsbruck-Austria-Innsbruck-1km-km.osm.
Generating Report(s).
Report(s) generated for Austria-Innsbruck-Austria-Innsbruck-1km-km.osm - Gene
```

Figure D.9: OSM-CAT application window (Analysis Finished)


Minimum Longitude -6.6019546

Minimum Latitude 53.3750057

Maximum Longitude -6.5800004

Maximum Latitude 53.3881335

Figure D.10: Latitude & Longitude entry window (Showing default values)

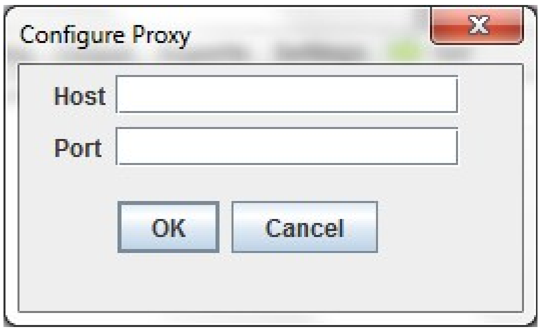


Figure D.11: Configure Proxy entry window

Appendix E: Eclipse Development Setup Guide

The following is a short guide designed to help potential developers set up their environment, in preparation for developing OSM-CAT. The complete source code, and all required libraries can be downloaded from our web server [44].

1. Download and install the Eclipse Integrated Development Environment (IDE) [47].
2. Download and install the Apache Maven [48] software project management and comprehension tool. Consult the Maven Setup guide for correct setup instructions.
3. Download the OSM-CAT source code directory from our web server.
4. Open Eclipse and import an *Existing Maven Project*;
    - *File->Import->Maven->Existing Maven Project*
5. Download the Required Libraries from our web server.
6. Setup the project *Build Path*
    - In Eclipse, right-click the project directory in the *Package Explorer*
    - *Build Path->Configure Build Path*
    - In the *Properties* window, open the *Libraries* tab
    - Click *Add External Jars,* and choose the following JAR files from the Required Libraries directory
        - *commons-logging-1.1.1*
        - *itextpdf-5.1.2*
        - *jcommon-1.0.16*
        - *jfreechart-1.0.13*
        - *josm-latest*
        - *jts-1.8*
    - Still in the *Libraries* tab, click *Add Class Folder,* and add the following directories from the OSM-CAT project directory
        - *externalclasses*
        - *resources*

Upon completing the above steps, your environment should be configured correctly, and you can now start developing OSM-CAT.