BIRKBECK COLLEGE, UNIVERSITY OF LONDON

A Cycle Route Analysis System for London

MSc Computer Science project report, Department of Computer Science and Information Systems, Birkbeck College, University of London 2016

Johannes P.N. Neethling

9/19/2016

Supervisor: Nigel Martin

This report is substantially the result of my own work, expressed in my own words, except where explicitly indicated in the text. I give my permission for it to be submitted to the JISC Plagiarism Detection Service.

Contents

List of figures

Abstract

This report describes the design and implementation of a cycling route analysis system. The system consists of a simple modular design architecture with a risk modelling component as well as a route analysis and visualisation component. The risk vectors include traffic accident risk and exposure to air-pollution. The traffic accident risk model involved the implementation of a snap to line algorithm in combination with proximity analysis. The air-pollution risk model involved the implementation of a spatial interpolation algorithm. Route analysis includes distance measurement, slope gradient and exposure to risk. The system also includes dynamic, interactive cartographic visualisations. The development work was done using the Geotools GIS development libraries in combination with the Java programming language.

1. Introduction

Cycling in an urban environment presents several cost factors, as well as risk factors, that can be modelled and used in the evaluation of routes. The cost factors may include the physical distance and slope gradient of the route while the risk factors may include the probability of traffic accidents and exposure to air-pollution, both of which are considered high in London. The primary objective of this project was to design and implement a software system that can evaluate travelling routes against such criteria and produce visualisations and reports of the results. The aim is to allow users to make informed decisions regarding the routes they use in terms of safety and efficiency. A secondary objective was to implement a multi-objective, optimised shortest route finder to further contribute to the above aim.

This project report is divided into 6 further sections. A summary of the background research is presented in section 2 which briefly covers the required concepts and data structures before moving to an overview of algorithms and methods that are useful in the modelling and analysis of such datasets. Section 3 covers the specifications including aims and objectives, requirements and development methodology. This is followed by the system design in section 4, which includes a description of the public data services, I/O formats, architectures and technologies used. Section 5 deals with the implementation of the application. This includes the realisation of design patterns, construction of the risk models, processing, analysis and visualisation of the route data and the graphic user interface. System testing is covered in section 6 which deals with both unit testing of software components and validation of models and results. This is followed by a critical evaluation of the project and a discussion of the scope for further work in section 7.

The report contains several appendixes. Appendix 1 provides a list of acronyms and abbreviations used, appendix 2 contains the UML class diagrams for the system, appendixes 3 and 4 provides listings of relevant code fragments for the implementation of algorithms and validation of models respectively while appendix 5 contains listings of those nodes of the Project Object Model which shows all the dependencies of the software.

2. Summary of background research

2.1 Concepts and data structures

2.1.1 Coordinate reference systems

Any useful computer representation of a geographic feature, such as a route, needs to be geo-referenced. This is the purpose of a Coordinate Reference System (CRS). Care has to be taken when combining data sources that are based on different underlying systems to prevent inaccurate visualisations and measurements.

Data points captured by Global Positioning System (GPS) are always in a geographic CRS which represents coordinate pairs for points on the earth's surface in angles measured from the equator (giving the latitude) and the prime meridian (giving the longitude). Web Mapping Services (WMS) such as Google Maps and Open Street Map are projected for display on a flat screen using a mathematical transformation of coordinates, leading in most cases to a so-called pseudo Mercator CRS. Any such projection must cause some distortion in the size, shape or relative position of the features represented and may lead to inconsistent distance measurements (Huisman and de By, 2009, p.223).

However in order to visually overlay data points from a GPS device (or features derived from such data points) onto a WMS backdrop, the GPS data must undergo an equivalent transformation (Topf and Hormann, 2015). This transformation would then render the data in question in a CRS that is inappropriate for accurate measurements. It proved essential to take these factors into account during the implementation phase of this project and to do multiple on-demand transformations in order to support both accurate measurements and sensible visual display, whenever each was required.

2.1.2 Computer representations of geographic information

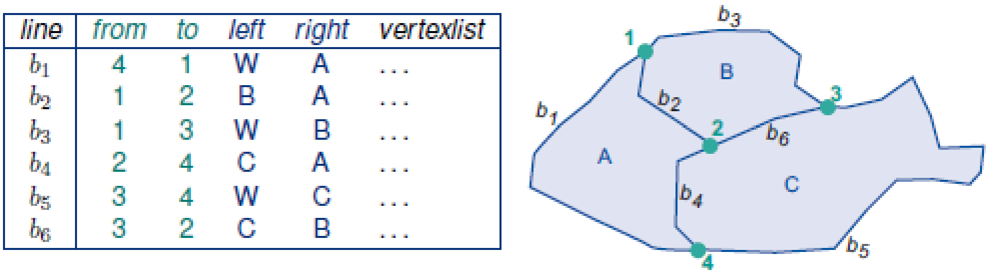
Spatial data is usually represented in computer memory as either raster tessellations or vector data. Raster data is best suited to represent phenomena for which a value can be measured or interpolated at any point in the study area. These types can be seen as continuous phenomena or geographic fields (Huisman and de By, 2009, p. 69). Examples include topography, often by way of a digital elevation model (DEM) and air-pollution. The vector data model can be used for phenomena that occur at specific locations with well-defined boundaries such as districts, streets and buildings, called geographic objects (Huisman and de By, 2009, p. 70).

Huisman and de By (2009, p.85) describe a tessellation as "…a partitioning of space into mutually exclusive cells that together make up the complete study space. With each cell, some (thematic) value is associated to characterise that part of space…In all regular tessellations, the cells are of the same shape and size, and the field attribute value assigned to a cell is associated with the entire area occupied by the cell." The smaller the individual cells, the higher the spatial resolution at the cost of an increase in data volume. Geo-referencing of raster data is achieved by providing a reference for one of the corners.

During the implementation of this project, a limited amount of use was made of raster data for a purely analytic purpose. However, the notion that it should be possible to query the value of a specific cell, given a location, was exploited during the implementation phase of this project in order to calculate the slope gradient of a route. More fine-grained analysis requires the use of vector data and it is important that the technology used in such implementations supports predictable, theoretically sound geographical abstractions. This project relied heavily on vector analysis and therefore it makes sense to provide a brief review of those theoretical requirements.

The vector model requires that every piece of spatial data be geo-referenced with coordinates. The simplest vector is a point defined by a single coordinate pair (x, y). Each point has an identifier so that it may be referred to as a component in a more complex structure or related to thematic attributes when used on its own. Lines are composed of sets of points, which in this context are called nodes and vertices. Each line (edge or arc) is defined by the two end nodes, and zero or more internal vertices. These define the individual, approximated, straight components of the feature called segments (Huisman and de By, 2009, p.96). The more individual vertices (and segments) included in the line, the better the approximation of the feature will be at the cost of an increase in the data volume. Interconnected lines can form a network with topology rules to indicate aspects such as connectivity, capacity, cost of traversal etc.

A more complex vector representation is that of area features. Huisman and de By (2009, p.98) explains that "…each area feature is represented by some arc/node structure that determine a polygon as the area's boundary". The boundary model is often used to store features in situations where adjacent polygons share a boundary, in order to avoid data redundancy and minimise computational complexity. Each polygon's boundary is divided into non-overlapping arcs with an indication of which polygon is to the left and which to the right of each arc (Huisman and de By, 2009 p. 99).



line	from	to	left	right	vertexlist
b_1	4	1	W	A	. . .
b_2	1	2	B	A	. . .
b_3	1	3	W	B	. . .
b_4	2	4	C	A	. . .
b_5	3	4	W	C	. . .
b_6	3	2	C	B	. . .

Figure 1: A simple boundary model for the polygons A, B and C. For each arc we store the start and end node as well as a vertex list (omitted for brevity) and its adjacent polygons. The outside world polygon is denoted by W (Huisman and de By, 2009, p.100).

All forms of vector data should have an identifier to relate extra thematic data with the feature being represented. In cases where complex structures such as lines and polygons are used to represent features it is useful to understand how their composite parts are employed in the construction of the feature in order to facilitate the design of algorithms that may require iteration over those composite parts.

2.2 Algorithms and methods

2.2.1 Spatial analysis

Spatial analysis can refer to operations on raster or vector data. As the modelling and analysis components of this project relied almost exclusively on vector data, the rest of this discussion will focus on vector analysis only.

Spatial analysis on vector data includes proximity analysis, spatial joins and overlay operations all of which were used in this project in either implementation or validation of models and analytic results. Proximity analysis is the simplest form of vector analysis. It involves calculating a buffer at a specified distance around a feature as a new polygon output (Trodd, 2005 p.2). These may then be used in spatial joins and/or overlay operations.

Spatial join operations are focused on the selection of existing features based on topological relationships between features in two input datasets (Jacox and Samet, 2007, p.3). They can be seen as queries that use location to link two relations and can be used, for example, to find all line segments (perhaps representing roads) that intersect polygons in another dataset (perhaps representing accident hot-spots) and then associating the attributes of those polygons with the relevant lines.

Overlay operations are used to define new features in an output dataset based on spatial joins. Normally one of the input datasets would contain polygon features, but the other may contain polygons, lines or points. All are based on the manipulation of coordinate sets.

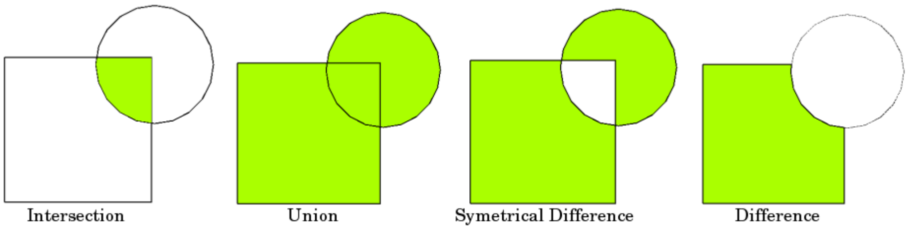


Figure 2: Some overlay operations on vector data (using polygons in this case).

2.2.2 Spatial data interpolation

Interpolation techniques are used when there is a requirement to assign values to all possible points in the study area based solely on data from sample points, which may be randomly distributed. There are several well understood techniques available to achieve this but some require considerable domain knowledge regarding the functioning of the geographic field in question and/or may produce an output in the form of a specific vector representation with a narrow applicability. This project used inverse distance weighting (IDW) which, as a generic technique was found to be appropriate for producing the required air-pollution risk model.

The IDW equation to predict a value at each point in the study area can be expressed as follows:

$$\sum_{i=1}^{n} \frac{m_i}{d_i^p} \Big/ \sum_{i=1}^{n} \frac{1}{d_i^p}$$

Where:

- n is the number of measurements selected
- m is the measurement value at point i
- d is the distance between prediction point and measurement point
- p is the power to be applied

It is common to make p=2, i.e. using the squared distance as weighting factor and to use a “window” that iterates over the dataset (Huisman and de By 2009, p.333).

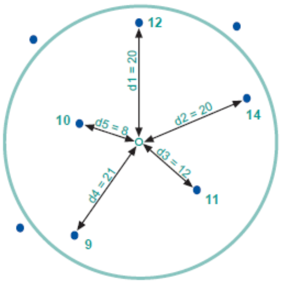


Figure 3: IDW with a circular window centred on the target cell, showing distances to measurement points and their values (Huisman and de By 2009, p.334).

2.2.3 Optimal path and multi-objective optimisation algorithms

The family of algorithms in this category can be described as network analysis. Investigation into them was necessitated by the secondary objective of this project. They are widely used to solve geographical problems (such as vehicle routing) and also information and communication technology problems. Optimal path finding techniques are used to find the least-cost path (a sequence of lines) between two nodes (origin and destination) in a network (Huisman and de By 2009, p.417). Perhaps the most famous is the algorithm devised by Dijkstra in the 1950s to find the optimal route from a source to all possible destinations (a fact also exploited by the network layer of the TCP/IP stack to generate routing tables).

Dijkstra’s algorithm aims to satisfy a single objective: to minimise the cost (distance) of traversal. There is often a requirement to satisfy more than one objective and in such cases the focus changes from finding the shortest path to finding a set of compromise solutions (Ehrgott et al, 2012 p. 656). The additional objectives may be more nebulous, such as risk (accident risk, pollution exposure etc.). In this case, a modelling component has to be added as risk factors, for example, can be expressed as either total risk value or a maximum risk value on any path (Galbrun et al, 2016 p. 164).

3. Specifications

3.1 Aims and objectives

The aim of this project was to build a system that can process and evaluate cycling routes and can produce reports and visualisations of the results in a way that is easily understandable to the user.

The primary objective was to provide functionality for users to specify routes so that the system can evaluate them in terms of distance, changes in elevation and risk (traffic accidents and exposure to air pollution). Delivering this functionality required the design and implementation of a model to locate and quantify risk, a set of procedures for processing route data and executing the spatial analysis needed to evaluate routes - both in terms of the route's inherent characteristics and in relation to risk factors - and finally the means to present a suitable visualisation of the results.

The secondary objective was to provide functionality to compute a small set of optimised paths that represent a reasonable trade-off between safety, distance and efficiency. Delivering this functionality required the availability of a suitable graph representation which could incorporate multiple cost (and risk) factors and the implementation of a path finding algorithm. All the paths must be shown and appropriately symbolised on a map, together with metrics, for comparison.

3.2 System requirements

A number of requirements were originally identified for the proposed system. These have subsequently been refined in the light of lessons learned from test-running individual components of the system. The functional requirements which form the basis of the specification are listed below. All the requirements apply to the primary and secondary objectives of the system, except for I, which applies only to the secondary objective:

A. The system must ingest and process road safety and air pollution data and then use it to automatically build the required risk models which in turn must be stored for later utilisation with regards to the evaluation and/or computation of routes.
B. Requirement A must be repeatable, on demand, so that the risk models can be rebuilt whenever new road safety and air pollution data becomes available.
C. The system must allow the user to choose which version of the risk models should be used in any given computation – in particular for air pollution which may have a significant temporal variance over the course of hours, days and seasons.
D. The system must allow the user to choose an appropriate DEM for route slope gradient computations.
E. The system must present the user with a method to specify routes or upload routes for evaluation.
F. The system must do any spatial analytical computations required to evaluate routes in terms of their inherent cost characteristics and the identified risk factors.
G. The system must present the results of the evaluations in the form of a map depicting the location and magnitude of risk in relation to the route using colour coding and symbols.

H. The system must present the results of the evaluations in the form of a textual report that contains the relevant computed metrics.
I. The system must compute and display alternative, optimised routes based on the start- and end-points of user specified routes.

3.3 Development methodology

This project adopted a phased development approach with two major versions coinciding with the primary and secondary objectives identified in section 3.1. The major versions were further divided into minor versions which each delivered additional functionality in succession. The reason for adopting this approach was mainly due to the lack of familiarity with the required technology. In this regard, Tegarden et al (2013, p.19) states that: “Phased development–based methodologies are good…because they create opportunities to investigate the technology in some depth before the design is complete”. In effect each minor version had its own analysis, design and implementation phases allowing the detailed design to evolve as the project progressed.

The original project schedule was designed in a way that would guarantee the availability of a working product by the due date, even if it had to be a cut-down version of the original specification. The fall-back position was to drop the secondary objective and associated specifications and only produce version 1 of the system if time pressure dictated such a decision. However, since the design evolved, opportunities also arose to introduce different ways of meeting certain requirements and to include functionality at a different stage of development than initially planned. Because of these facts it is deemed more appropriate to discuss the actual versions produced, and the functionality that each delivered only after covering the final design in the next section.

4. Design

4.1 Input data

The system requirements identified in section 3.2 necessitated the availability of several different sources of input data. These are now briefly discussed here with more details on how exactly they were used in the next section covering the implementation.

4.1.1 Background cartography

Background map data is required for visualisation and can be downloaded and stored in a data tier or accessed via a public web mapping service (WMS). The main disadvantage of the former option is that all the data has to be symbolised and styled up for display in the system, which is a time-consuming cartographic function, not a computer science exercise. This project therefore uses a publically available WMS for background map visualisation.

Google Maps is probably the most famous WMS. It offers a useful set of APIs for embedding the service into third party applications, but the data is proprietary and is therefore not freely consumable in an internal or OEM application without incurring a commercial overhead (https://developers.google.com/maps/). The logical alternative was to use crowd-sourced data published by the Open Street Map (OSM) Foundation (https://www.openstreetmap.org).

Although the source vector data is freely downloadable, at the time of writing the OSM Foundation did not offer an API for directly embedding the service into a bespoke desktop application. There are, however several third party organisations that offer such a service, based on their in-house styles, free of charge (http://wiki.openstreetmap.org/wiki/WMS). The system that is the subject of this report was designed to offer the end-user the opportunity to connect to one of these services in order to render a backdrop map for visualisation.

4.1.2 Risk data

Road safety data is published by the Department for Transport (https://data.gov.uk/dataset/road-accidents-safety-data). It is available for download as CSV files that identify the location of each accident and its severity along with several other fields of potential interest. The data is captured by police officers who attend the scene of the accident and record the required details as part of their normal duties. In most cases, a GPS device is used to capture the location of the incident. The data is sent on to the Department for Transport, where it is collated and published on an annual basis. This means that the data can be obtained once per year and used to build a risk model that would be re-usable until new data becomes available.

Air-pollution data is available from the London Air Quality Network website hosted by King's College (http://www.londonair.org.uk/LondonAir/API/). It is delivered as XML and JSON feeds containing hourly updates from automated measuring stations with known locations distributed across London. This means the data can be obtained automatically at runtime and used to build a near up to date risk model on an hourly basis. At the same time, older risk models can be retained and still be used for route evaluation in case they are considered to be particularly representative of conditions at a certain hour of the day, weekday or season.

4.1.3 Reference data

Since the accident data is captured with a GPS device, it is clear that this dataset must contain a significant number of inaccurate positions. Positional accuracy is typically between 5 and 10 meters, 95% of the time depending on factors such as atmospheric conditions (Ordnance Survey, 2016). In order to ensure that the accidents fall exactly on actual roads when building the traffic accident risk model, it was necessary to have access to a geographically accurate vector model of the road network so that the captured accident locations could be "snapped" onto legitimate road segments. A further requirement for this dataset was as a basis for a path finding algorithm. The OSM roads dataset was chosen as it includes dedicated cycling routes, tow paths and other pedestrian paths. Furthermore it is the same underlying data that the WMS use and as such would align perfectly with the provided visualisations. The dataset can be exported from http://www.openstreetmap.org.

The implementation of the air-pollution risk model also required a reference dataset. The building of the model took a somewhat unconventional approach in that no raster was produced during the interpolation of data from the measuring stations. If a raster was used, it would have been necessary to eventually extract vectors from it for spatial analysis and visualisation. This process is near impossible to fully automate as the levels of generalisation required depends heavily on the range of actual values obtained and cannot be reliably fixed in advance.

Instead a vector-based reference grid was constructed beforehand, which could act as "scaffolding" for the model. This reference grid can be re-used whenever a new air-pollution model needs to be constructed, something that may in theory occur on an hourly basis.

Both these two reference datasets are in the open-source Shapefile format from Environmental Systems Research Institute (ESRI). Shapefiles actually consist of a set of files with a common prefix that are meant to work together from the same folder location. At a minimum there must be a .shp file containing all the geometry elements, a .dbf file containing the attribute records (that map one-to-one to geometry elements) and a .shx file for indexing (ESRI, 1997). Many applications require several other supplementary files to be present, most notably the .prj file containing the native CRS and projection parameters of the geometries.

4.1.4 Topography

The digital elevation models from the environmental agency were originally considered as a source of topography data (http://environment.data.gov.uk/ds/survey/). However these have an extremely fine-grained spatial resolution and it was found that the data volumes involved are excessive and impractical for the project. The Ordnance Survey has recently made its OS Terrain 50 product available under the open data scheme. It is available as a raster DEM with a spatial resolution of 50 meters which makes it ideal for the use case in question and therefore chosen as the recommended topography dataset for the system. It can be downloaded from https://www.ordnancesurvey.co.uk/business-and-government/products/terrain-50.html.

It should be noted though that any DEM can be used in the system as long as it covers the required geographical area and is in a raster format supported by the technology stack of the project. These include ASCII grid, Geo-Tiff and JPEG2000 to name just a few.

4.1.5 User route data

The system was designed to accept user route data at runtime. Two options were originally considered, namely heads-up input and file upload. The former option would have required functionality for the user to indicate a path by clicking on a map interface similar to what is available in Google Earth and other desktop GIS applications. This was considered to be too difficult to implement in the available time and so the latter option was chosen. The system therefore only allows the user to upload routes that were pre-encoded in appropriate files. The chosen format for these files is KML, a well-documented XML dialect which is relatively easy to produce and parse. The simplest way to produce such files is by exporting a route created in Google Earth. Another way to create route files is to use a mobile application that can capture a route while it is being traversed. Route files can also be constructed, albeit laboriously, using an ordinary text editor - if the route coordinates are known. The only preconditions are that the KML files contain at least one route (as opposed to just points and/or polygons) and that the file validate successfully against the official KML XSD (http://www.opengeospatial.org/standards/kml).

4.2 System architecture

The original intention was to implement the common 3-tier architecture with a light-weight desktop client, an application logic tier in the middle and a data tier running PostgreSQL at the back. This was partly based on the assumption that the spatial analysis capabilities of the PostGIS database extension would be a crucial element in delivering the required functionality. However more extensive testing of the Geotools library for Java (discussed below) revealed that all the required functionality could in fact be realised by interaction with file based resources and in-memory structures. For example, it was discovered that the air-pollution input stream could be turned into a DOM and used to construct the risk model in main memory without the intermediate step of storing the XML data from the feed on disk. Furthermore it was discovered that the geographical abstractions and associated methods provided by this library would be able to support the full range of spatial analytical requirements specified.

Since a good argument still existed for keeping risk model construction separate from analysis and visualisation (refer to requirements A, B and C), it was decided to structure the system as two independent software components that each run in its own process. Figure 4 shows a modular view of the system design with the arrows indicating the flow of data between input and output, via the components.

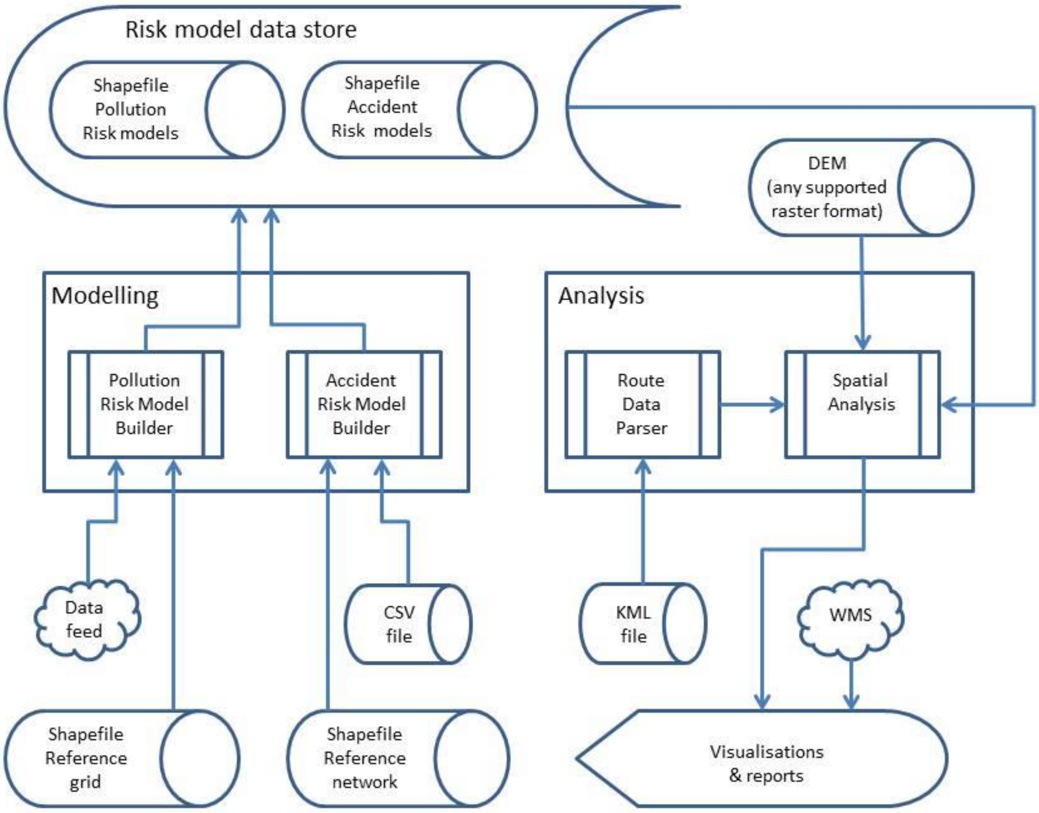

Risk model data store
Shapefile Pollution Risk models
Shapefile Accident Risk models
DEM (any supported raster format)
Modelling
Pollution Risk Model Builder
Accident Risk Model Builder
Analysis
Route Data Parser
Spatial Analysis
Data feed
CSV file
KML file
WMS
Shapefile Reference grid
Shapefile Reference network
Visualisations & reports

Figure 4: A modular view of the system design.

It should be noted that the two main components can work independently from each other. For example, there is no hard requirement for the risk models to be built by the modelling component. If appropriately structured risk models can be had by some other means (say by constructing them using an off-the-shelf GIS application) then those would be usable in the analysis component as well. This means the analysis component can actually work for any conceivable study area on earth. The modelling component, on the other hand, is designed to take in data sources that are very specific to organisations in the United Kingdom and to convert those sources into appropriate risk models for London in particular.

4.3 Technology

On account of the limited previous development experience of the author, only two programming languages were considered for the system namely Python and Java. Python was a strong candidate. As a dynamically typed, interpreted language it has an intuitive learning curve, while offering many third party libraries for dealing with geographic abstractions (https://pypi.python.org/pypi/GDAL/), CRS transformations and geodetic calculations (https://pypi.python.org/pypi/pyproj) and spatial analysis (https://pypi.python.org/pypi/Shapely) to name just a few. However, such an approach would have led to many inter-related, version specific dependencies as each of the required libraries would have had to be installed on top of the core Python language.

Instead the Java specific Geotools GIS toolkit was selected for all development work related to this project (http://www.geotools.org/). By using Geotools, no libraries needed to be pre-installed (along with the overhead of managing dependencies) before commencing development work. It relies on the Apache Maven software project management tool (https://maven.apache.org/) to specify all the required dependencies in a so-called project object model (POM). This is essentially an XML based manifest that allows the system to automatically pull in the required Java Archive (JAR) files on demand. Maven also controls the software build process via the POM, allowing the final products to be easily packaged as self-contained executable binaries. A final consideration was the ease, generally speaking, with which a Maven based project can be set up and configured in an integrated development environment for Java which, in the case of this project, was the Mars version of the open source Eclipse IDE.

Geotools wraps the basic vector geometries of the individual points, lines and polygons in several layers of abstraction. In the case of linear vectors, for example, the individual geometries are handled as an array of segments (as per section 2.1.2) wrapped together in a combined geometry object. Such geometry objects together with any associated attributes are wrapped in a feature. Features, on their part are wrapped in a feature collection. The feature collection can be used to construct a feature layer with an associated style for display on a map or a feature source for committing to disk via a file data store - to specify encodings for various formats such as ESRI Shapefiles, KML, etc. Obviously this process is often followed in reverse, going from data on disk to getting access to individual geometries. To facilitate this, a special iterator is provided for feature collections to allow the streaming of data from disk instead of loading it all at once into main memory.

Geotools provides a range of interfaces and classes for working with geographic data. It uses the Java Topology Suite (JTS) as the basis for spatial analysis (including proximity analysis and overlay operations). The latter can also be used, in combination with a special filter interface, to implement spatial joins (refer section 2.2.1). Filters can also be used for aggregate functions such as finding the sum or maximum value of an attribute in a feature collection. There is a special helper class to deal with CRS definitions and mathematical transformations. It also provides some useful classes for complex analysis of the geometries such as the geodetic calculator. This can be used to perform computations against the underlying ellipsoid model, for example to find the true distance (along the earth's curvature) between two points. In addition, there is a graph interface that can be used in conjunction with several traversal classes, including Dijkstra and A-star shortest path finders.

In summary, it appears that the Geotools technology caters for a wide range of GIS related use cases. However the high levels of complexity inherent to the GIS data do tend to complicate the application thereof. The challenge of using this technology is further exacerbated by the relative scarcity of tutorials and examples, especially where more advanced use cases are concerned.

5. Implementation

5.1 Implementation of design patterns

The Model-View-Controller (MVC) design pattern was used for both the modelling- and analysis/visualisation components of the systems. As such they both work in essentially the same way when viewed from a high level. They both have a GUI defined in a view class containing certain controls, mainly in the form of standard Java buttons. The views have public methods to add event listeners to the buttons. These methods are called from the controller class with the listener object passed in as a parameter.

The listeners themselves are inner classes of the controller in order for them to have access to the private members of the parent class. The listener classes make calls to both the view (to update it) and the model (to get data from it). All exceptions thrown by the model are caught and handled by the controller class. Any long running processes (such as the construction of the risk modelling I/O resources) are executed in a Swing worker object to avoid the problems associated with running such tasks on the main event dispatch thread. The models (in the MVC context) make calls to a number of custom made classes where the main algorithms reside. These are discussed in more detail in the following sections. The focus of the discussion falls on the workings of the custom-made algorithms as opposed to standard method calls on the provided Geotools library objects, although these are obviously mentioned where appropriate. Also see Appendix 2 for UML class diagrams of the components.

5.2 Construction of the risk models

The discussion of the implemented algorithms below follows a sequential approach with clearly demarcated steps. The corresponding code fragments for each of these steps are given in appendix 3 for cross-referencing.

5.2.1 Traffic accident risk model

The construction of the traffic risk model relies on the raw accident data in CSV format (slightly over 140,000 records for the 2015 country-wide dataset) and an ESRI Shapefile of the road network of London (around 203,000 line segments). The process of constructing the model consists of four sequential steps namely taking in the text from the CSV file to create coordinate points for the accidents, snapping the points to the line segments of the road network, transforming the snapped points to the correct CRS and finally constructing buffers around the points.

Step 1:

The first step is to read the accident locations from the CSV file into an array of JTS coordinates using a custom object called from the MVC model class. This object filters the records of the CSV file to include only the accidents of interest, i.e. those in London ("Police_Force" = 1 indicating the Met police) and those of sufficient severity ("Accident_Severity" < 3 indicating fatal and serious accidents) in the model.

Step 2:

Next the geometries of the line features from the road network are placed into a spatial index. A loop is then created over the accident points obtained in step 1. A fixed sized search envelope is placed around each accident point and this is used to query the spatial index of lines. This query returns a list of location indexed lines that are in range for snapping that point to. Now another (inner) loop is set up over these lines and, for each one a custom snapping object is created. This object evaluates the lines, in turn, to identify the best line (if any) to snap the accident point to. Furthermore, a JTS indexed line object has a method to project out the coordinates of the vertex on that particular line segment that is closest to a reference point. This is exploited by the custom snapping object to find the appropriate vertex of the line to snap the accident points to. The snapped point is returned by the snapping object and is used to construct a new point feature, which is then placed in a new feature collection representing the set of corrected points.

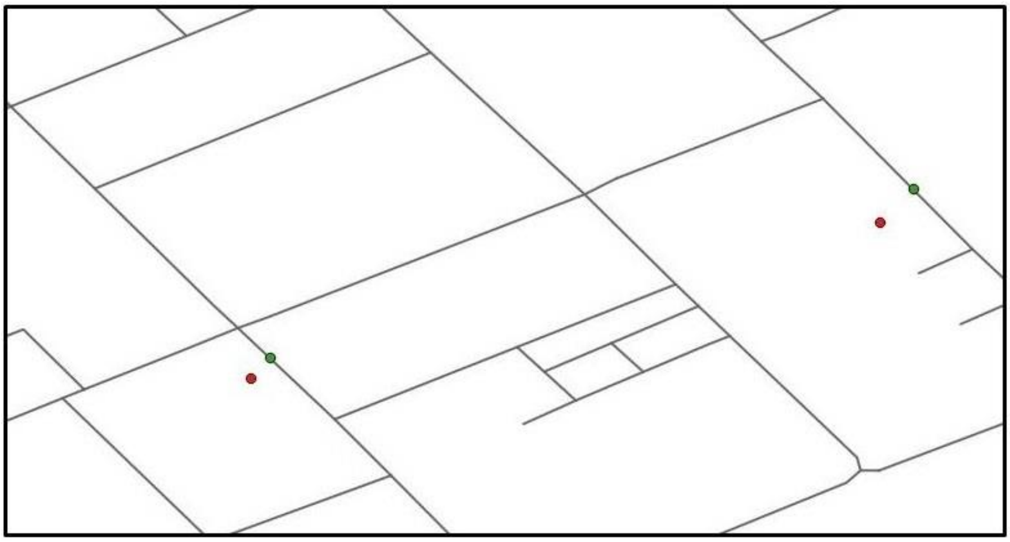


Figure 5: Some results from the snapping algorithm with original points in red and corrected points in green.

Step 3:

To transform these points into the appropriate CRS, a math transform function is set up using the helper class mentioned in section 4.3. The geometries of each feature in the new point collection (obtained from step 2) are then transformed to the pseudo Mercator projection used by the WMS. The reason the points are first transformed before setting up the buffers is explained in section 2.1.1. In short, if the buffer polygons are created first and are then transformed, they become elongated ovals (as opposed to perfectly circular) with obvious implications for both spatial analysis and visualisation.

Step 4:

The buffer polygons are now created with a radius of 25 meters using the appropriate call to each point geometry object. These polygons are then wrapped in features and added to a feature collection. The latter is used to create a feature source and committed to disk via the Shapefile data store object with a file name and location specified by the user, thus producing a simple traffic accident risk model for analysis and visualisation.

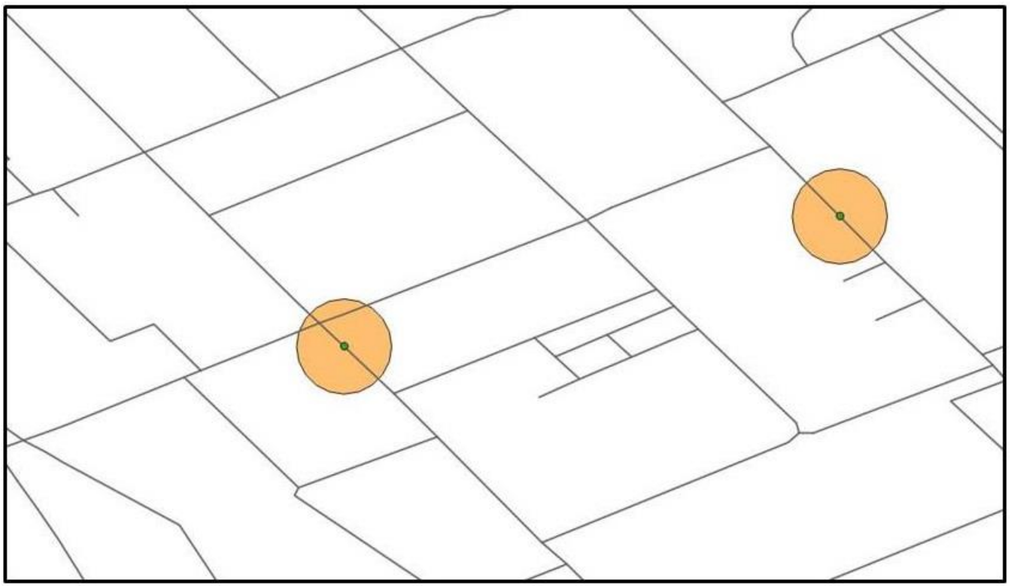


Figure 6: Some buffer polygons from a completed accident risk model.

5.2.2 Air-pollution risk model

The construction of the air-pollution risk model relies on an ESRI Shapefile containing a reference grid (11,000 polygons in the case of the dataset constructed for this project) and a URL connection to the raw pollution data feed (containing 100 XML elements at the time of writing, each representing a measurement site). Theoretically, the reference dataset can contain polygons of any shape and size as long as it is organized in a structure containing regularly spaced vector polygons with a value field compatible with the double data type (initially null) which can be populated with interpolation values. The spatial extents of the grid must also cover the Greater London area and already be in the pseudo Mercator CRS. It is clear that smaller polygons will facilitate finer-grained analysis at the expense of higher computing overhead.

The chosen structure of the pre-constructed grid was hexagonal, with the distance between the opposing *sides* of each hexagon's north-south dimension at 1000 meters and the distance between opposing *corners* in the east-west dimension at around 1150 meters.

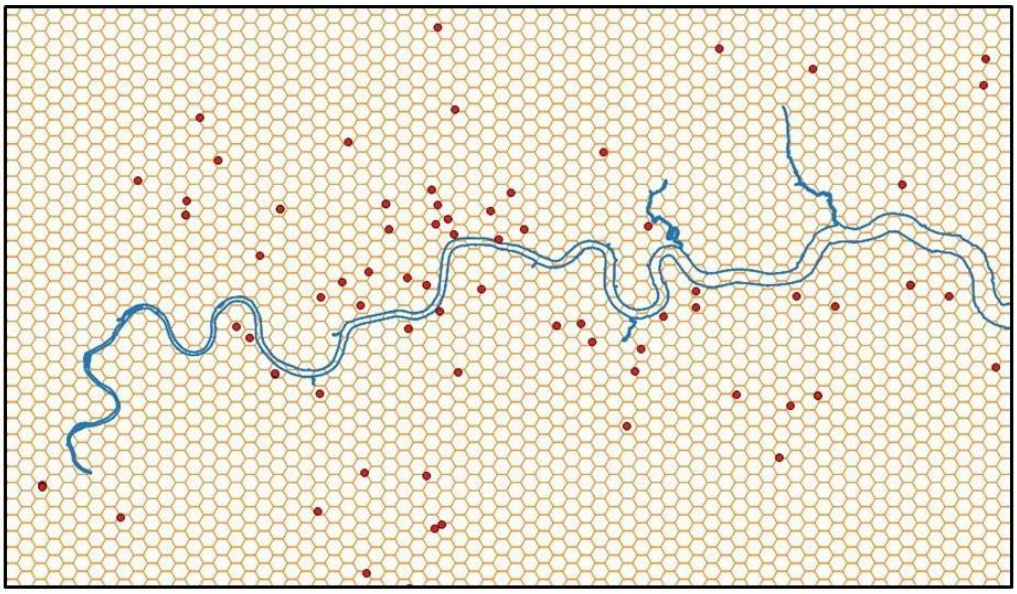


Figure 7: A section of the reference grid showing the shape of polygons with the location of some pollution measuring stations and the high-water lines of the river Thames for visual reference.

The process of constructing the air-pollution is divided into 3 steps. These include obtaining measurement points from the feed data, calculating the IDW interpolation and writing these results back to the value field of the reference polygons and finally filtering the polygons for inclusion in the model.

Step1:

The first step in obtaining the measurement points is to open a URL connection and obtain an input stream for the feed data. The input stream is then used to build a W3C DOM in memory. Once this is done, the input stream is closed. The appropriate nodes are then accessed and placed in a node list using the methods that comes with the DOM. The nodes in question are the second level child elements of the document representing the measurement sites and containing the latitude and longitude amongst other attributes. These nodes are sent over to a custom object that turn them into point features. A for-loop is used in this custom object to access the child nodes of the sites which contain the measurements for each type of pollution particulate (so-called species).

```
<LocalAuthority LocalAuthorityName="Barking and Dagenham" >
        <Site SiteCode="BG1" Latitude="51.563752" Longitude="0.177891" >
                <Species SpeciesCode="NO2" AirQualityIndex="1"/>
                <Species SpeciesCode="SO2" AirQualityIndex="1"/>
        </Site>
        <Site SiteCode="BG2" Latitude="51.529389" Longitude="0.132857" >
                <Species SpeciesCode="NO2" AirQualityIndex="1"/>
                <Species SpeciesCode="PM10" AirQualityIndex="2"/>
        </Site>
</LocalAuthority>
```

Figure 8: An example of a node from the air-pollution feed (with some attributes omitted for brevity).

For this application a simple average of measurements over all species are calculated, to be used as input value for the IDW interpolation. The coordinates, together with the averaged values are used to set up point features which, in turn is placed in a feature collection.

Step 2a:
The next step starts with the extraction of the feature collection of polygons from the reference grid by setting up the iterator. For each polygon, a centroid is calculated using the appropriate call to the polygon's geometry object to act as an interpolation point. The interpolation point is then transformed to the geographic CRS for compatibility with the measurement points obtained from the data feed in step 1. This is done in a similar way as already described earlier for the traffic risk model construction. At this point a custom IDW object is created to actually calculate the interpolated value for each reference polygon (discussed in step 2b below). A new IDW object is created for each polygon in the iteration, meaning it is only responsible for calculating the interpolation value for that one polygon. Once calculated, the value is used to update the relevant attribute of that polygon in the reference grid.

Step 2b:
The custom IDW object receives both the target point itself and the collection of measurement points from the calling code described in step 2a. For convenience sake the target point is already wrapped in a geodetic calculator object (as the calculator's starting position). A numerator and a denominator (both initially zero) are also initialised. Next the IDW object loops over the measurement points setting them, in turn, as destination positions for the geodetic calculator object. The distance between each measurement point and the target point is now obtained using a call to the geodetic calculator. An "if" statement is used so that measurement points are only considered if they are closer than 5000 meters from the target point thus forming a circular moving window as described in section 2.2.2. If the measurement point satisfies the distance criteria, the average pollution value obtained from the XML feed in the first step is now extracted from the measurement point and used in the calculation of the numerator for this interpolation target point. This is done by incrementing the numerator by the value over the distance squared. The denominator is likewise incremented by 1 over the distance squared. After the loop over all the measurement points is completed, the final interpolation for the target point is obtained by numerator over denominator with appropriate protection against the possibility of a zero denominator (if there were no measurement points within 5000 meters of the target).

Step 3:
The final step in the process is to filter the polygons for inclusion in the final model based on the attribute values obtained by the IDW interpolation. For this the aggregate function described in section 4.3 is used in a calculation to find the cohort of polygons representing the highest air-pollution at that point in time. A cut-off value is calculated to identify polygons that fall within the top third of all values and these are added to a new feature collection which is then committed to disk using exactly the same process as already described for the traffic risk model.

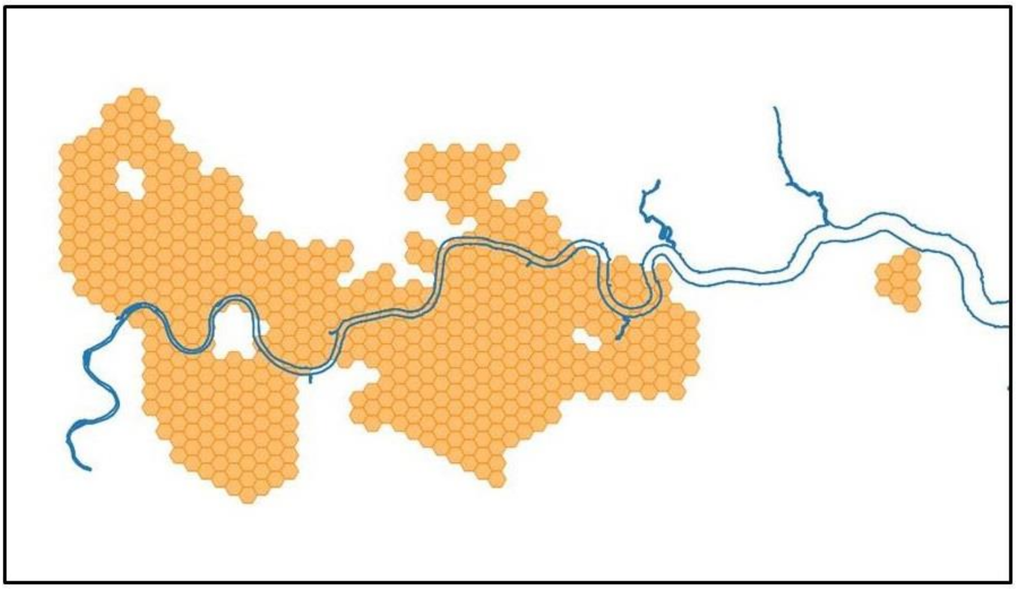


Figure 9: An example of a completed air-pollution risk model shown against the high-water lines of the river Thames for visual reference.

The discussed above described the algorithms of the modelling component of the system. The discussion now moves on to the analysis/visualisation component of the system.

5.3 Processing, analysis and visualisation of the route data

5.3.1 Processing of the route data

Processing the route data involves a number of steps namely parsing the input file and building a W3C DOM in memory, creating a line structure from the coordinates in the appropriate DOM node and finally wrapping them in a feature collection for inclusion in a map layer for analysis and display.

Although the Geotools library provides some KML readers, they were found cumbersome to work with and it was decided to write bespoke functionality for this purpose. As a result the KML is parsed as “vanilla” XML with appropriate checks to ensure the DOM complies with the requirements of the application. This is relatively easy since the KML schema requires the coordinates of a route to be in a single node as shown in figure 10.

```
<LineString>
        <coordinates>
                0.07068575887670292,51.49981150633879,0 0.07056216267570914,51.50000201203246,0
         </coordinates>
</LineString>
```

Figure 10: An example of a route node in a KML file (showing 2 coordinates with height values of 0, which is the default for routes captured in Google Earth).

The text data in the node are split - first at the white space and then at the comma - into text arrays, which are looped over to create an array of JTS coordinates (ignoring the height data). The next step is to create the line geometries using a call to the JTS geometry factory class. The geometries are then transformed to the correct CRS for display before being wrapped in a feature collection. As a final step, the feature collection is used to build a map layer with a default style for display.

5.3.2 Analysis of the route data

Analysis of the route data focussed on four metrics namely the length of the route, the average slope gradient of the route, the number of accident hotspots encountered on the route and the percentage of the route that runs through the most polluted areas. While the length of the route can obviously be calculated in isolation, obtaining the other metrics requires the loading of an accident risk model, an air-pollution risk model and a DEM (for calculating the slope gradient) into the analysis component of the application. This is achieved through standard Geotools calls to create geographical abstractions from the relevant pre-prepared files (whether created in the modelling component or obtained in some other way). The application will not allow analysis to commence before all the prerequisite resources are available.

Calculating the total length of the route is a simple matter of decomposing the route into its basic building blocks i.e. the individual straight line segments (as discussed in section 2.1.2) and iterating over those to get the length of each. This is achieved by getting the two defining vertices of the segment, transforming them back to a geographic CRS fit for accurate distance calculations and making a call to the geodetic calculator object. The slope gradient calculation involved getting the start and end point of the user route and obtaining the elevation from the DEM at those points. The DEM, being a raster type is abstracted to a so called grid coverage object with an evaluate method for obtaining values at specified points. Once the values are obtained, the calculation is simply to put the end height minus start height over route length multiplied by one hundred, thus giving the average slope gradient over the route.

Calculating the percentage of the route that is most polluted requires another loop over the feature collection of lines encapsulated in the user route. Next the geometry is extracted from each feature after which an iterator is set up for the polygons in the pollution risk model thus forming an inner loop in which the latter geometries are extracted as well. At this point a conditional test is done to check if the line geometry from the outer loop intersects the polygon geometry from the inner loop. If it does, the overlay operation is done (refer to section 2.2.1 and figure 3) using a call to the intersection method of the geometry object. It should be noted that these are two separate methods, namely intersect (which returns a Boolean) and intersection (which returns a new vector geometry). The lengths of the new line segments formed by the intersections are tallied to produce the distance that the route is subject to air-pollution (as modelled). After the iterators are closed, the final calculation can be completed. This simply entails the total of the polluted distance over the total distance of the route multiplied by one hundred, giving the polluted percentage.

Calculating the number of traffic accident hotspots (as represented by the buffer polygons in the risk model) on the route is done in much the same way as the air-pollution calculation described in the previous paragraph. Again a nested loop is used to test for intersects between lines and polygons, but this time the number of intersects evaluating to true are simply tallied to produce a value for the number of accidents hotspots on route. However care has to be taken not to count the same polygon many times over if there are a number of very short line segments intersecting the same polygon. For this reason the polygon's unique identifiers are place in a set of strings to avoid duplication. This is the structure of which the members are in fact counted to produce the final value. This method will also allow for the "selected" polygons and their attributes to be accessed at a later time for further analysis in some future version of the application.

5.3.3 GUI and visualisation of data

The modelling component provides no cartographic visualisations and therefore it relies on a standard Java Swing frame in its view class to render the GUI. The GUI consists of 4 Java buttons, a label conveying the current status and a progress bar. The two topmost buttons brings up file dialogs allowing the user to choose the reference dataset for each model. The two lower buttons executes the model construction. In the case of the traffic accident model, the system will prompt the user for an input file and then prompt for an output file path for the completed model. In the case of the air-pollution model there is only one prompt, for the output file path, as the input data comes from a pre-defined URL over the web. The lower buttons are each disabled until the appropriate reference dataset have been connected. All buttons are disabled while any risk model is being constructed. Since the construction of models run on separate threads it may have been possible to allow concurrent execution of these functions. However many Geotools classes, including the geodetic calculator, are not thread-safe and since the models typically takes less than a minute to construct it was deemed best to let the user wait this period out, while showing the progress. The progress is managed by a Java Beans property change listener in the model class reporting events to the Swing worker in the inner classes of the controller, which in turn updates the view.

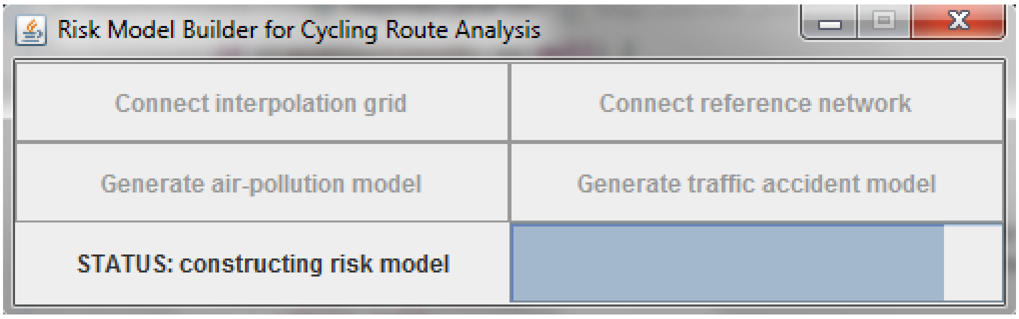
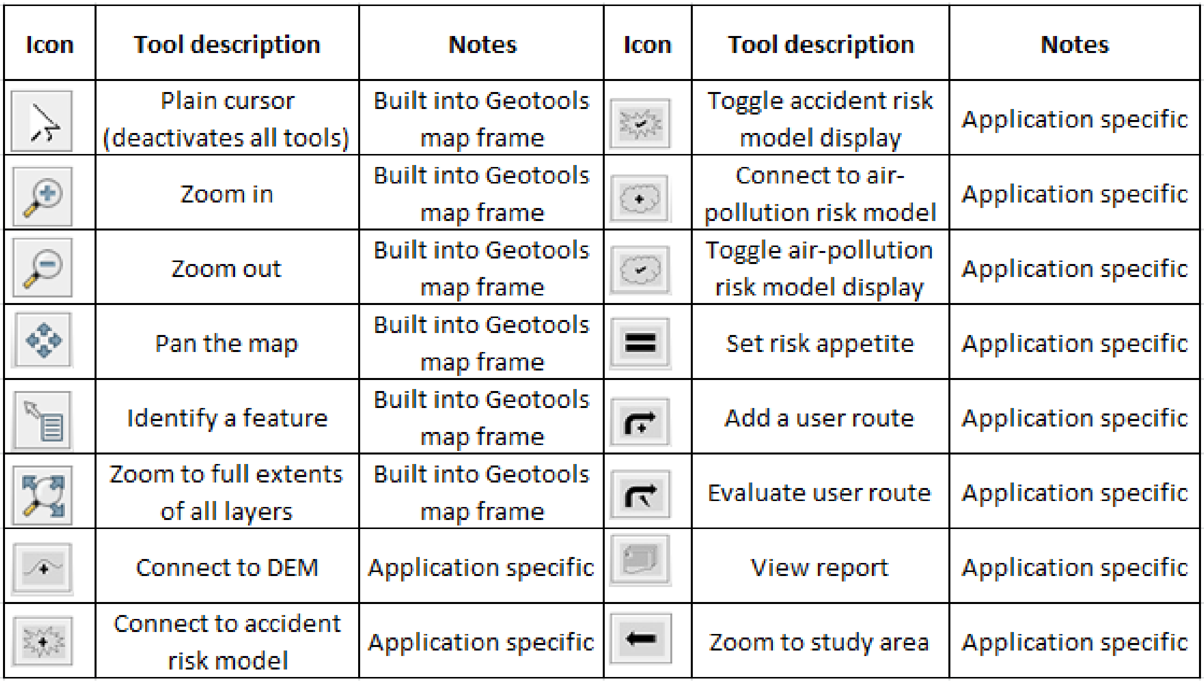

Risk Model Builder for Cycling Route Analysis
Connect interpolation grid
Connect reference network
Generate air-pollution model
Generate traffic accident model
STATUS: constructing risk model

Figure 11: The GUI of the modelling component shown mid-execution while constructing a risk model.

Both the GUI for- and the visualisation of risk- and route data are based on a Geotools map frame in the view class of the analysis/visualisation component, instead of a standard Java Swing frame. This map frame extends the standard Swing frame and therefore allows for much the same containers and controls to be included. It also allows for specialist containers such as the map pane which can be used to display cartographic content.

All spatial data is displayed on a backdrop map obtained from a WMS. A choice of 3 publically available services is presented at start-up (all using OSM data with bespoke styling), but the user is free to enter the URL of any suitable service. Once a service is selected, a so-called capabilities document is returned which is used to determine exactly how the service is to be used. This includes the choice of which layers of the service will be rendered as part of the backdrop map. All of this is handled by standard method calls on the provided Geotools library objects.

The GUI has 16 buttons in total. The first 6 buttons are built into the Geotools map frame and appears automatically when setting the enable toolbar value to true. The remaining 10 buttons are specific to the application and are added to the toolbar in the constructor of the view class. Figure 12 provides a summary of all the buttons in a table format.



Icon	Tool description	Notes	Icon	Tool description	Notes
↖	Plain cursor (deactivates all tools)	Built into Geotools map frame	✷	Toggle accident risk model display	Application specific
⊕	Zoom in	Built into Geotools map frame	☁	Connect to air-pollution risk model	Application specific
⊖	Zoom out	Built into Geotools map frame	☁	Toggle air-pollution risk model display	Application specific
✥	Pan the map	Built into Geotools map frame	=	Set risk appetite	Application specific
☰	Identify a feature	Built into Geotools map frame	↱	Add a user route	Application specific
⤢	Zoom to full extents of all layers	Built into Geotools map frame	↱	Evaluate user route	Application specific
⌒	Connect to DEM	Application specific	▭	View report	Application specific
✷	Connect to accident risk model	Application specific	←	Zoom to study area	Application specific

Figure 12: The buttons of the analysis/visualisation component.

The functionality of the built in buttons are self-explanatory and are common to most off-the-shelf GIS applications. The application specific buttons that control the connection to the DEM, accidents- and air-pollution risk models all present the user with a file dialog to enable them to specify the required resources. The two toggle buttons become enabled once the connections are made to the relevant resources and are simply used to hide (if currently shown) or show (if currently hidden) the risk models on the map display. The DEM is not actually displayed, but only used in the slope gradient calculation already discussed.

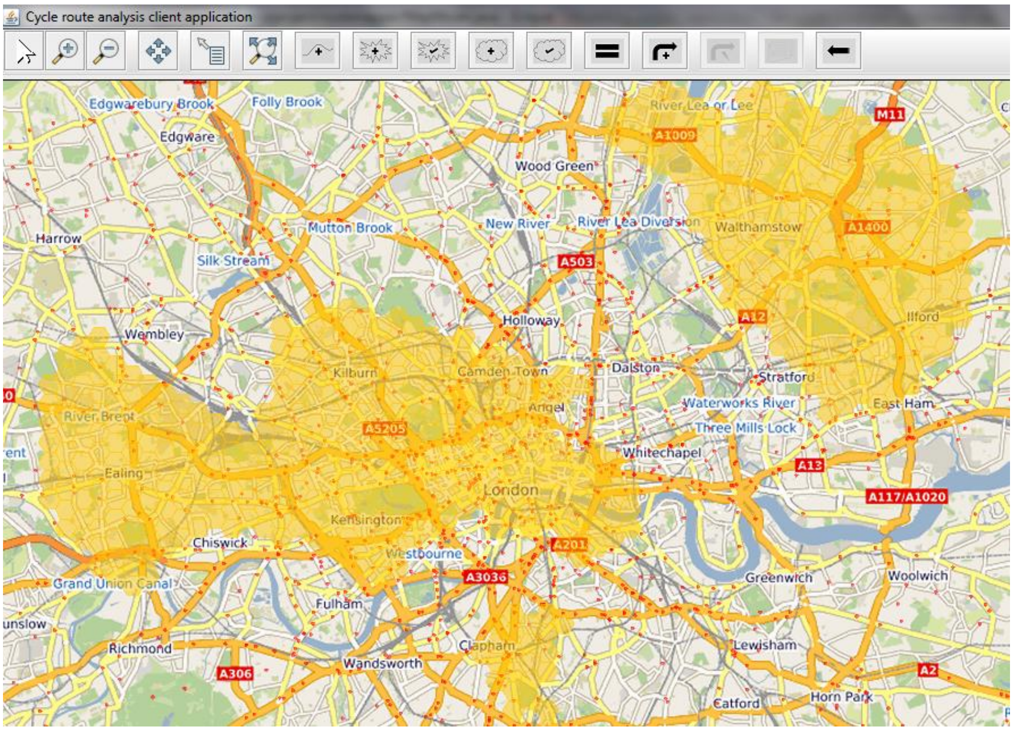


Figure 13: The map display showing the 2015 accident model (red dots at this scale) and an air-pollution model generated at a specific point in time (orange hexagons) against the preferred WMS backdrop.

The colours of the fill pattern and outline stroke with which the risk models are displayed are controlled by a style object, constructed by calling the static methods of the Geotools style factory class. The button for adding the user route also provides the familiar file dialog. After the selection of the KML file is made, the processing of the route data discussed in section 5.3.1 is executed. Once completed the map automatically zooms to the extents of the route and displays it in a defualt style.

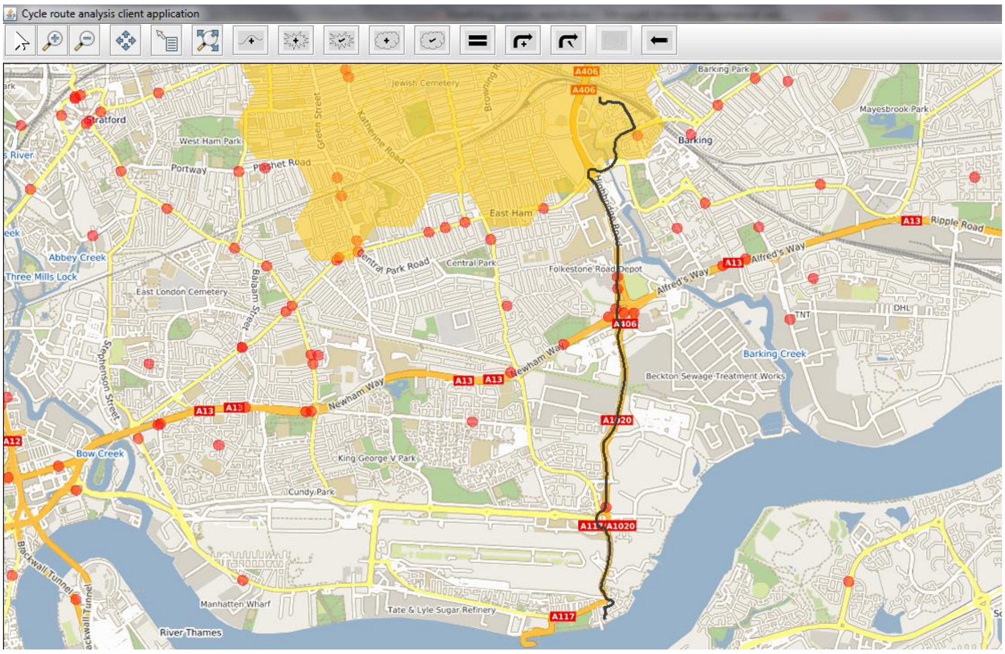


Figure 14: The map display showing the 2015 accident model (red polygons) and an air-pollution model (orange hexagons) with a user route in its defaults style against the preferred WMS backdrop.

Once all the required modelled resources and the route is available the evaluate button becomes enabled. The user can also click the risk appetite button at any point in time to change the maximum tolerated values for the two risk vectors from their default settings. The risk appetite settings are displayed in a Swing panel with two Java slider controls.

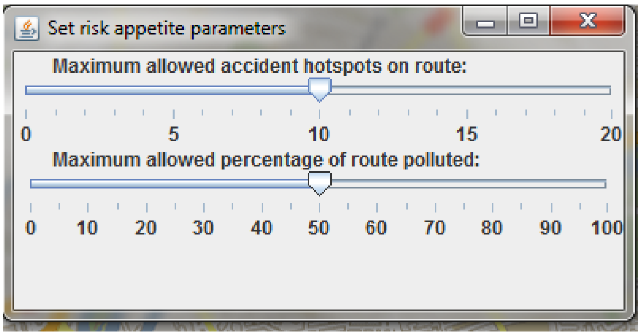
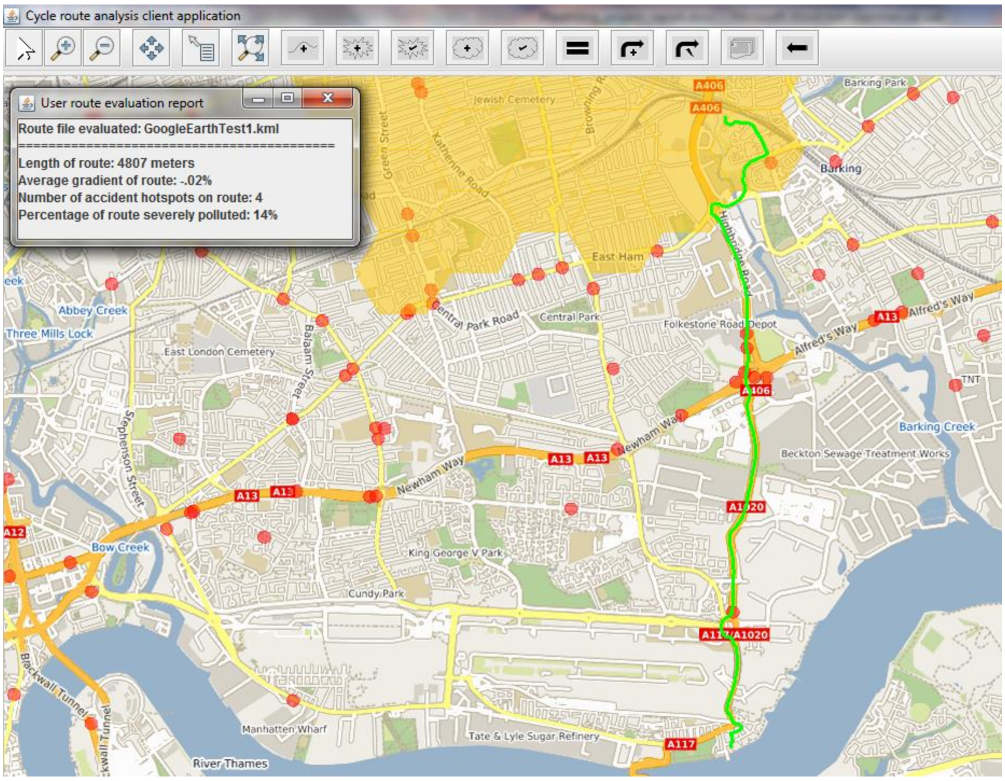

Set risk appetite parameters
Maximum allowed accident hotspots on route:
0
5
10
15
20
Maximum allowed percentage of route polluted:
0
10
20
30
40
50
60
70
80
90
100

Figure 15: The risk appetite controls.

When the user click the evaluate button all the metrics discussed in section 5.3.2 are calculated on Swing worker threads. The calculated values for the two main risk vectors are then compared to the risk appetite values and if any are greater than the maximum tolerable values then the route will turn red, else it turns green. This is achieved by resetting the style to one of two predefined styles using the relevant call to the map layer object.   Once the route is evaluated the reporting button is enabled. When this button is clicked another Swing panel is shown, which displays the calculated metrics. The user can add a different route, change to different models (both the DEM and the risk models), change the risk appetite settings and re-evaluate against the new setting at any point in time.


Cycle route analysis client application
User route evaluation report

```
Route file evaluated: GoogleEarthTest1.kml
================================================
Length of route: 4807 meters
Average gradient of route: -.02%
Number of accident hotspots on route: 4
Percentage of route severely polluted: 14%
```

Figure 16: The final visualisation of the evaluated route with a textual report of the calculated metrics.

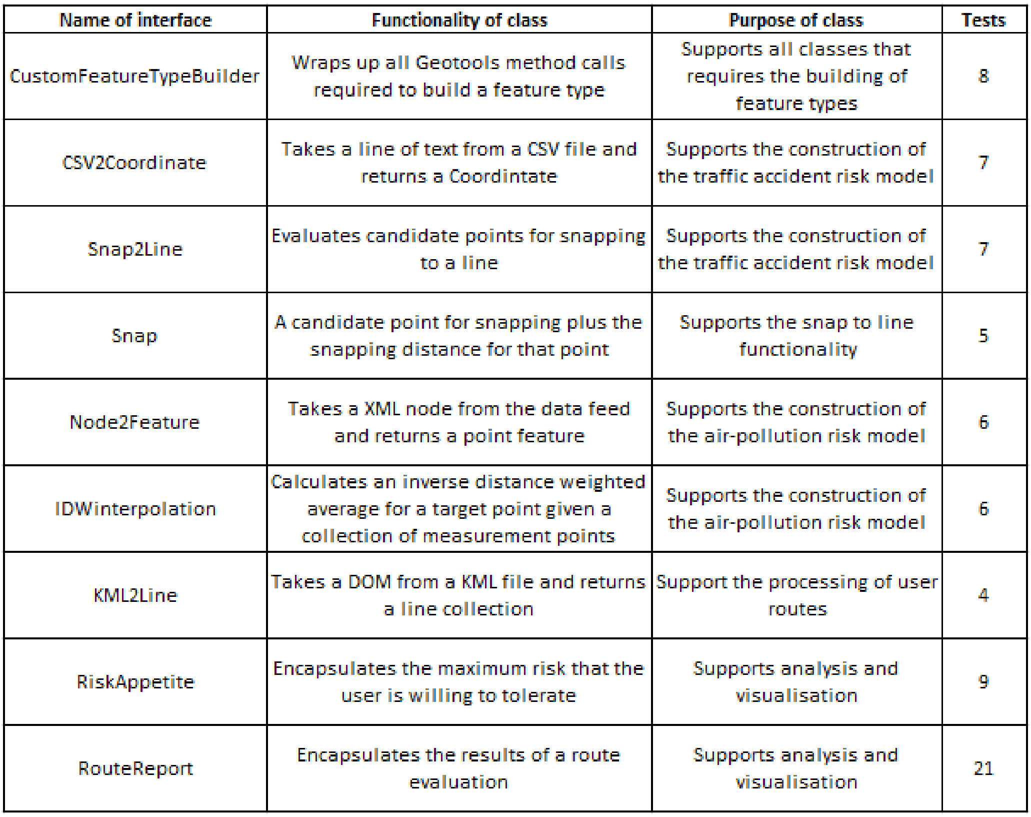
6. Testing

6.1 Unit testing

The unit testing for this project was done using version 4.11 of the Junit testing framework. The testing effort focussed on those classes that host the key algorithms and use custom-made objects rather than those that simply just call the methods of the classes in the Geotools library. As it turned out, the analysis/visualisation component could be implemented by mostly using rather straight-forward looping structures, together with calls to methods in the Geotools library that already provide the implementation of the required algorithms such as calculating distance, overlay and spatial join. The more complex development effort therefore went into the modelling component. The main examples of this, is the fact that there are no classes in Geotools that provide pre-implemented algorithms for IDW or snapping points to lines. The unit testing was therefore somewhat skewed toward the modelling component of the system. It is also true that some aspects of the modelling component also relied on simple calls to methods in standard out-of-the-box Geotools objects such as proximity analysis (i.e. buffering points for the accident model).

In total there are 9 classes that are subject to unit testing in this system. Of these, six are in the modelling component. Each of the test classes contain a battery of tests methods that aim to cover both normal execution paths and, at least the most obvious corner cases. Some of the test cases required extensive mocking-up of geographically abstracted objects. A simple example of this can be found in the snap to line tests, which required location-indexed lines as input. To mock these up required that the structures be built from scratch in the test itself. Building such structures involves a number of steps including constructing coordinates from primitive types, constructing linear vector geometries from the coordinates and finally indexing the lines. Some of the other test, including the IDW tests, required even more extensive mocking. This included building both feature types and the input features (that adheres to the type) from scratch. Partly because of the fact that several test classes needed the building of feature types and partly because several production classes required such a procedure as well, it was decided to wrap the Geotools method calls required to achieve this into a custom feature type builder class which is also subjected to its own battery of tests.

Since the unit test are integrated into the build procedure of the software project management framework, they are all run whenever a version of the software is packaged into executable binaries. The Maven framework ensures that the final packaging cannot be completed if any of the tests fail during the build process. This means that the development process could proceed with a high level of confidence that existing functionality would not be broken by the introduction of new functionality. Figure 17 summerises those classes subject to unit testing along with their functionality and overall purposes in the context of the system.



Name of interface	Functionality of class	Purpose of class	Tests
CustomFeatureTypeBuilder	Wraps up all Geotools method calls required to build a feature type	Supports all classes that requires the building of feature types	8
CSV2Coordinate	Takes a line of text from a CSV file and returns a Coordintate	Supports the construction of the traffic accident risk model	7
Snap2Line	Evaluates candidate points for snapping to a line	Supports the construction of the traffic accident risk model	7
Snap	A candidate point for snapping plus the snapping distance for that point	Supports the snap to line functionality	5
Node2Feature	Takes a XML node from the data feed and returns a point feature	Supports the construction of the air-pollution risk model	6
IDWinterpolation	Calculates an inverse distance weighted average for a target point given a collection of measurement points	Supports the construction of the air-pollution risk model	6
KML2Line	Takes a DOM from a KML file and returns a line collection	Support the processing of user routes	4
RiskAppetite	Encapsulates the maximum risk that the user is willing to tolerate	Supports analysis and visualisation	9
RouteReport	Encapsulates the results of a route evaluation	Supports analysis and visualisation	21

Figure 17: Summery of unit tests.

6.2 Validation of models and analysis results

The validation of models and analysis results required the use of a professional grade desktop GIS application. The most powerful examples of such applications are probably the Feature Manipulation Engine (FME) from Safe Software (https://www.safe.com/fme/fme-desktop/) and the ArcGIS suite from ESRI (http://www.esri.com/software/arcgis/arcgis-for-desktop). Both of these are commercial offerings aimed at high-turnover production houses and therefore not really in the scope of a project of this nature. Instead a reasonably powerful open-source equivalent was sought out. The decision was made to use QGIS (http://www.qgis.org/en/site/about/index.html) on account of the author's previous experience with this application.

6.2.1 Validation of the accident model

Two validation tests were conducted for the accident risk model. For the first test the input CSV file was first opened in Microsoft Excel and a filter applied to the fields used in the application to identify an accident location and severity value. The number of records that were selected in Excel was 1973. Next the output model was loaded into QGIS and the option selected to show the feature count. QGIS reported the presence of 1973 features.

For the second validation test the centroids of the polygon buffers of the accident risk model were extracted in QGIS. The OSM roads were also loaded, first ensuring that they were projected into the correct CRS, using QGIS functionality. Since QGIS do not have a true geodetic calculator, all the data had to be explicitly transformed to a projected CRS (British National Grid) that supports accurate distance measurements. A very narrow buffer of 0.01 meters (in the real world) was placed around the roads. Now the QGIS topology checker was used to define a rule for validating these features. This rule was that the centroids must fall within this buffer distance. If it does, then one can safely consider these points to be exactly on the lines representing the roads. After running the test QGIS found no errors. To validate the test itself a control polygon was made and a similar rule defined for that polygon. As expected QGIS found 1973 errors after running the control test.

6.2.2 Validation of the air-pollution model

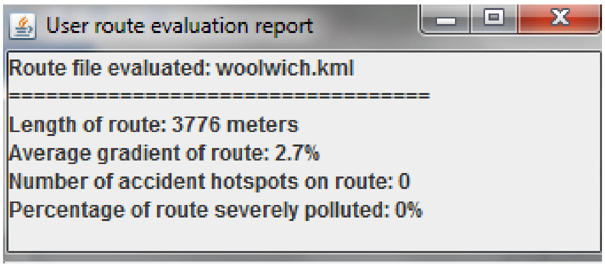
One validation test was conducted for the air-pollution risk model. As a first step the feed data was saved to disk and converted to a CSV file. This was done using a simple Java class and a Python script of which the code listings are given in appendix 4. This CSV file was then uploaded into QGIS.

Now the model builder component was used to build an air-pollution model that represents the same set of measurements. This was then loaded into QGIS after which the centroids of the individual hexagons were extracted while preserving the interpolation attribute. These centroids were then transformed to the appropriate CRS using QGIS functionality as described above. Three of the centroids were randomly selected for validation. Next a buffer of 5000 meters was placed around the selected points to identify the measurements that were supposed to contribute to the interpolation at that point. The distance to each point was determined by using the QGIS straight line measurement tool. Finally a manual calculation was done which verified the accuracy of the sampled interpolation values.

6.2.3 Validation of analysis results

The validation of analysis results were split into two parts. To validate the route length and slope gradient metrics, any test route would be appropriate as long as that route does not run over perfectly flat terrain. For the accident count and percentage of route polluted one does require a test route that actually crosses accident hotspots and polluted areas.

For the first part, a route was captured in Google Earth that runs through an area which is known to have significant variation in topography. This route was then evaluated in the application with the report shown below in figure 18.

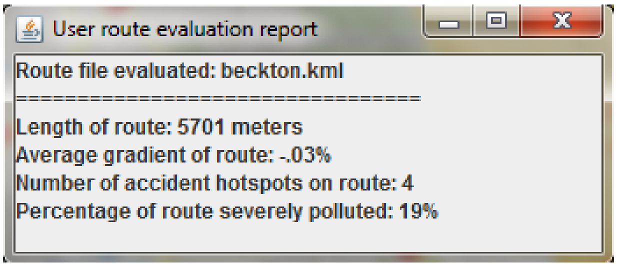


```
Route file evaluated: woolwich.kml
==================================
Length of route: 3776 meters
Average gradient of route: 2.7%
Number of accident hotspots on route: 0
Percentage of route severely polluted: 0%
```

Figure 18: Evaluation report for test route woolwich.kml.

This same route was now loaded into QGIS and the field calculator used to compute the length, again ensuring an explicit projection of the data was done before completing the procedure. The output from this calculation was equal to what was produced by the application, taking into account normal rounding behaviour. Next the DEM was projected and loaded into QGIS and queried at the start and end points of the route. A manual slope gradient calculation was done, which gave the expected result of 2.7%.

For the second part of the validation, a route was captured in Google Earth that is known to intersect some accident hotspots and modelled pollution. This route was then evaluated in the application with the report shown below in figure 19.



```
Route file evaluated: beckton.kml
================================
Length of route: 5701 meters
Average gradient of route: -.03%
Number of accident hotspots on route: 4
Percentage of route severely polluted: 19%
```

Figure 19: Evaluation report for test route beckton.kml.

Now this route was loaded into QGIS together with the accident model. To validate the accident count metric, a spatial join between the accident polygons and the route was done. For this procedure, it was acceptable to allow QGIS to do an automatic on-the-fly projection between the route data (still in the geographic CRS which is the default for Google Earth KML) and the accident model (in pseudo Mercator). In QGIS, this procedure produces an output dataset which has to be queried to find features on which attributes were actually joined. The outcome of that query returned 4 features, as expected.

For the final validation test, both the route and the pollution hexagons had to be explicitly transformed to the projected CRS. The intersection of the route with the pollution hexagons were calculated next and the length of that part computed as described earlier. Finally a manual calculation was done to validate the polluted percentage at 19%.

7. Evaluation

7.1 Critical evaluation

7.1.1 Requirements

All the requirements associated with the primary objective of this project were met in full. Meeting these requirements presented a significant challenge as it involved the evaluation of multiple criteria, based on heterogeneous data sources. This included the consideration of near real-time data and highly abstracted geographical constructs. It also included the provisioning of simple, but effective cartographic visualisations. To support these requirements it was necessary to develop custom risk models that included challenging algorithms such as snapping points to lines and IDW interpolation structures.

The requirement associated with the secondary objective was not met. An attempt was made to implement path finding functionality, based on the Dijkstra traversal class of the Geotools library, but this could not be completed in the available time and was subsequently left out of the final implementation. As a result the system offers no functionality for calculating alternative, optimised routes from any given origin and destination points. The completion of this requirement will therefore be considered under the scope for further work along with other aspects that may be improved or built upon.

7.1.2 Technology

The choice of using Geotools over a combination of other alternatives such as Python or even JavaScript proved to be reasonably successful even though the technology was almost entirely unknown to the author at the start of the project. The fact that Geotools were specifically designed to work with Java meant that it provided an ideal launch pad for getting the project off the ground quickly and smoothly. It also provided the opportunity for using open source data services such as OSM instead of propriety data such as Google Maps. This in turn opens other possibilities for further development into areas such as OEM applications that would not have been possible with propriety data.

7.1.3 Methodology

The phased development methodology that was followed allowed considerable leeway for experimentation with the Geotools technology at the outset, and a continued learning curve throughout the project. The main disadvantage of the approach was that the fast and continuously evolving design meant that it was not entirely possible to follow a strict test driven design approach from the outset. This led to a situation where large swathes of code were not easily testable and considerable effort had to be taken to refactor that code at a late stage of the project. This involved a large amount of retrospective abstracting away of functionality to get to a point where key algorithms could be tested in isolation. As such, the code coverage of testing remains a relative weakness of this project.

7.1.4 Architecture

Some compromise was required in terms system architecture as well. For the main use case that was envisioned for the system, it would have been a better scenario to provide multiple tiers so that the responsibility for sourcing and managing input data is removed from the end user and placed under the control of a system administrator. However, significant additional effort would have been required to provide serializable wrappers for the geographic abstractions that would have needed to be passed around between the middle tier and the client tier in a RMI scenario, for example. The decision was therefore made that the extra effort, especially in the light of the severe time limit, would not be worth doing for a project of this nature.

7.2 Scope for future work

7.2.1 Multi-objective optimised route computation

The logical starting point for future work is to complete the requirement associated with the secondary objective. This, in itself would require a number of tasks to be completed. In the first instance, some additional research would need to be conducted on how to get a suitable graph object from the OSM roads dataset. Secondly, some additional research and/or experimentation might be required to confirm which of the Geotools traversal classes (Dijkstra or A-Star), if any would be most suitable for this particular use case. This is particularly important considering the size and complexity of the graph in question. Thirdly it would be necessary to find a method for transferring the risk models onto the graph object for path finding and then to see if the Geotools traversal classes can accommodate optimisations based on multiple objectives. If not, a final step would be to develop a custom algorithm for this purpose.

7.2.2 Risk models and metrics

A second area for potential future work is to improve the risk models, and following that to expand the analysis beyond the four simple analytical metrics considered in this project. Although the models were never meant to be scientifically robust from a traffic management or environmental science perspective, it would be useful to investigate alternative methods of producing more robust models.

In terms of the accident model it would have been useful to have a better statistical understanding of which recent historical traffic incidents are most indicative of actual future risk. The complete accidents dataset contains a vast collection of fields, in different relations that were not utilised in this project. These include the vehicle types involved, the time of day and the weather conditions to name just a few. The only field that was actually used (apart from location) was the severity of the accident and it was only used to determine inclusion into the model or not. The inclusion of more attributes into the model would also mean that a more comprehensive analysis could be supported. This in turn would mean that more detailed metrics could be produced, such as a weighted accident score for a route based on multiple measures. Alternatively this metric could include a breakdown of the types of accident hotspots encountered on the route.

Future work could also aim to refine the air-pollution model. As it stands, the model represents an indication of the most affected area at the time of that particular set of measurements. This means that the result of analysis is greatly affected by temporal patterns that is not statistically normalised by the system. For example, even if the overall level of pollution happens to be low at the time of model generation, the model will still include the top third of all interpolated values. This may cause a situation where the air-pollution risk is either under- or overestimated in any given model.

There are at least two approaches that could be followed to improve this situation. Firstly the calculation of the metric could be adapted to take the actual interpolated value (which is available in the relevant attribute of each polygon of the model) into account. Instead of simply giving a percentage of the route that runs through the pollution affected areas (as modelled) the metrics could then take actual interpolated values into account as part of some sort of pollution score for that route. A second, perhaps more robust approach would be to adapt the model itself to include all values over a specific, immutable cut-off value for each species of pollutant. These cut-off values could be based on real health advice to make the model more relevant for real world situations.

Another aspect of route analysis that could be improved in future is that of the slope gradient. Instead of just providing a metric indicating the average gradient of the entire route, a visualisation could be provided in the form of a route profile, perhaps in the form of a line graph. This would be entirely achievable with the data and DEM already available to the system.

7.2.2 System architecture

A final area identified for possible future work is that of system architecture. A multi-tier architecture would better facilitate the continuity of data and the possibility of more robust risk models. One example of this is the inclusion of multiple years of accident data, integrated into a back-end database with finer control over other attributes, including vehicle type. The provisioning of an application- and/or web server would also open the possibility of many types of client applications, including browsers and mobile applications potentially connecting to the system.

8. Conclusion

The aim of this project was to build a system that could process and evaluate cycling routes and could produce reports and visualisations of the results in a way that would be easily understandable to the user.

The project demonstrated significant technical challenges, but these could for the most part be met by using the Geotools GIS development libraries in combination with the Java programming language. The nature of the project also necessitated a thorough theoretical grounding in order to better understand the workings of the geographical abstractions required to meet the challenge.

The project was conducted according to a reasonably well-defined approach, from research through design, implementation and evaluation. While certain problems could not be fully addressed in the available time, the project does manage to demonstrate many further possibilities that could be achieved by using this technology stack.

References

Ehrgott, M, Wang, J.Y.T, Raith, A, and van Houtte, C. 2012. A bi-objective cyclist route choice model. *Transportation Research Part A: Policy and Practice,* [e-journal] 46(4) Available through: Birkbeck University of London Library website <http://www.bbk.ac.uk/lib/elib/> [Accessed 6 March 2016].

ESRI. 1997. *ESRI Shapefile Technical Description* [PDF]. Environmental Systems Research Institute Inc. Available at: <https://www.esri.com/library/whitepapers/pdfs/shapefile.pdf> [Accessed 19 August 2016].

Galbrun, E, Pelechrinis, K and Terzi, E. 2016. Urban navigation beyond shortest route -The case of safe paths. *Information Systems*, [e-journal] 57 Available through: Birkbeck University of London Library website <http://www.bbk.ac.uk/lib/elib/> [Accessed 6 March 2016].

Huisman, O. and de By, R.A. eds. 2009. *Principles of Geographic Information Systems.* Enschede, The Netherlands: The International Institute for Geo-Information Science and Earth Observation.

Jacox, E.H. and Samet, H. 2007. Spatial Join Techniques. *ACM Transactions on Database Systems*, [e-journal] 32(1). Available through: Birkbeck University of London Library website <http://www.bbk.ac.uk/lib/elib/> [Accessed 28 February 2016].

Ordnance Survey. 2016. *Accuracies using a single GPS receiver* [Online]. Ordnance Survey. Available at: <https://www.ordnancesurvey.co.uk/business-and-government/help-and-support/navigation-technology/gps-beginners-guide.html > [Accessed 19 August 2016].

Tegarden, D, Dennis, A, Wixom, B.H. 2013. *Systems Analysis and Design with UML*. Singapore: John Wiley & Sons.

Topf, J. and Hormann, C. 2015. *Projections/Spatial Reference Systems* [Online]. OpenStreetMapData. Available at: <http://openstreetmapdata.com/info/projections> [Accessed 7 April 2016].

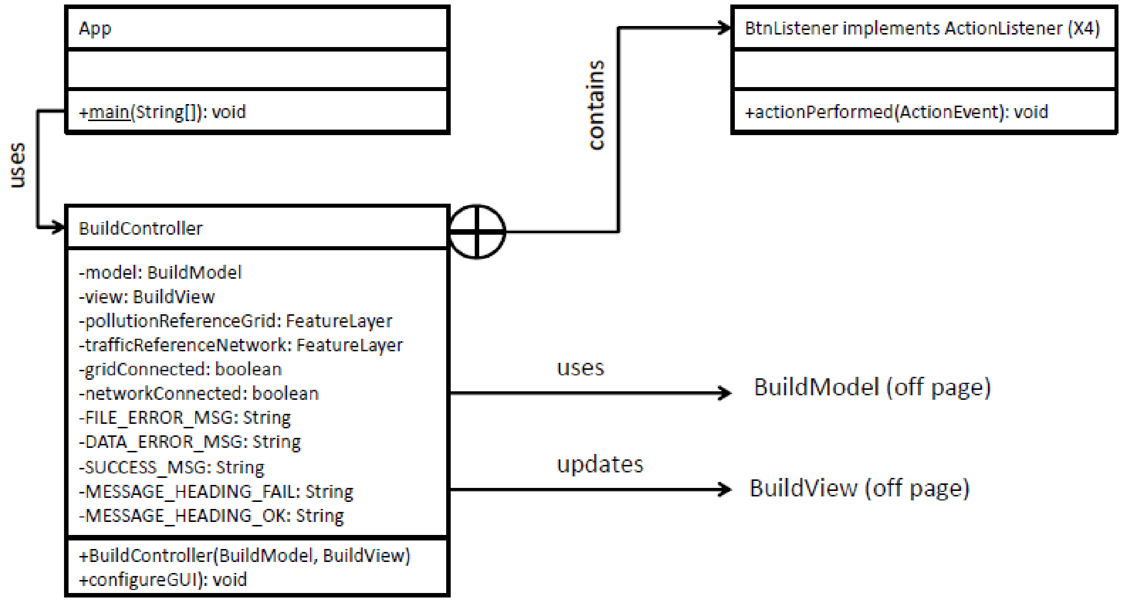
Trodd, N. 2005. *Proximity analysis* [PDF]. gisknowledge.net. Available at: <http://gisknowledge.net/topic/spatial_operations/trodd_proximity_analysis_05.pdf> [Accessed 28 February 2016].

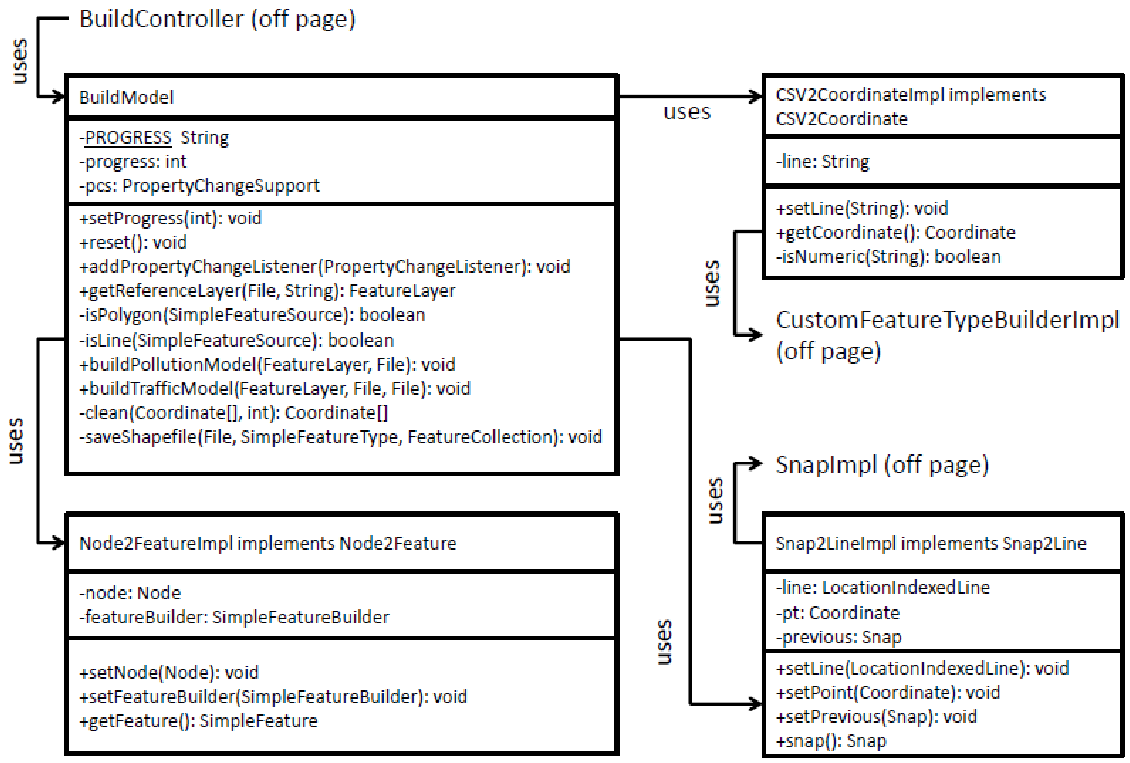
Appendix 1: List of abbreviations and acronyms

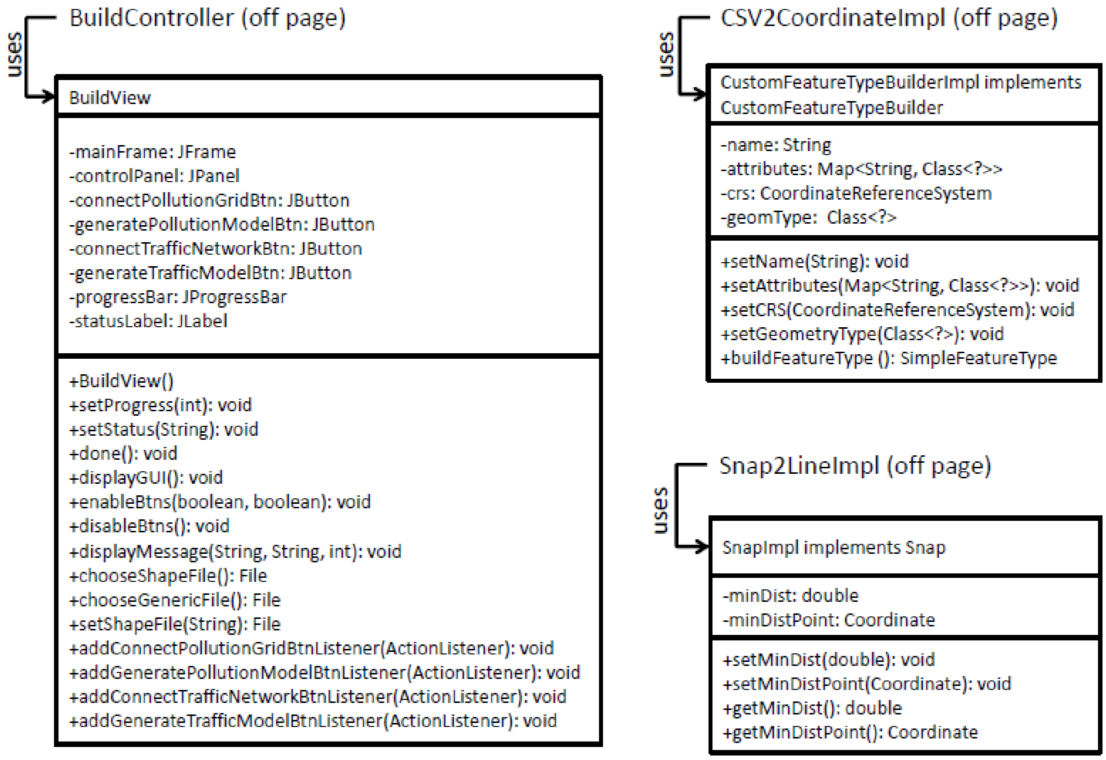
API	Application Programming Interface
ASCII	American Standard Code for Information Interchange
BNG	British National Grid
CRS	Coordinate Reference System
CSV	Comman Seperated Values
DEM	Digital Elevation Model
DOM	Document Object Model
ESRI	Environmental System Research Institute
GIS	Geographic Information System
GPS	Global Positioning System
GUI	Graphic User Interface
I/O	Input/Output
IDE	Integrated Development Envrironment
IDW	Inverse Distance Weighting
JAR	Java Archive
JPEG	Joint Photographic Experts Group
JSON	JavaScript Object Notation
JTS	Java Topology Suite
KML	Keyhole Markup Language
MVC	Model-View-Controller
OEM	Original Equipment Manafucturer
OS	Ordnance Survey
OSM	Open Street Map
POM	Project Object Model
RMI	Remote Method Invocation
TIFF	Tagged Image File Format
UML	Universal Modelling Language
URL	Universal Resource Locator
W3C	World Wide Web Consortium
WMS	Web Mapping Service
XML	Extensible Markup Language
XSD	XML Schema Definition

Appendix 2: UML Class diagrams

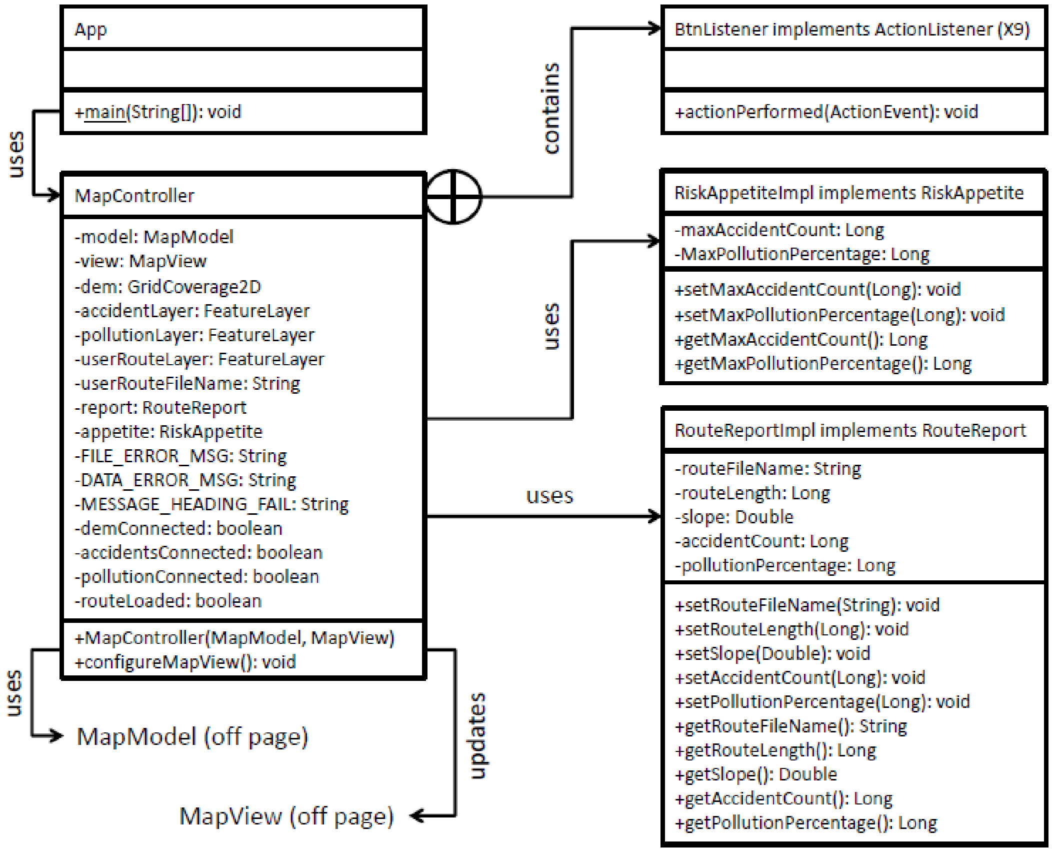
Modelling component

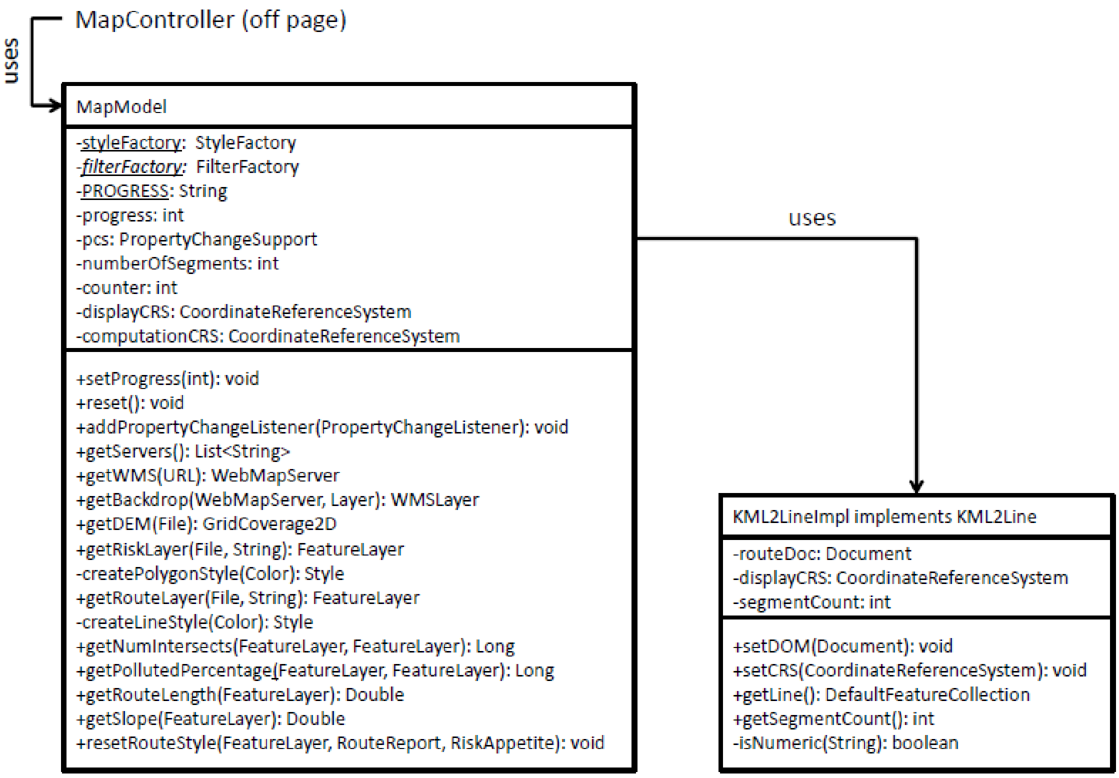

App
+main(String[]): void
uses
BuildController
-model: BuildModel
-view: BuildView
-pollutionReferenceGrid: FeatureLayer
-trafficReferenceNetwork: FeatureLayer
-gridConnected: boolean
-networkConnected: boolean
-FILE_ERROR_MSG: String
-DATA_ERROR_MSG: String
-SUCCESS_MSG: String
-MESSAGE_HEADING_FAIL: String
-MESSAGE_HEADING_OK: String
+BuildController(BuildModel, BuildView)
+configureGUI): void
contains
BtnListener implements ActionListener (X4)
+actionPerformed(ActionEvent): void
uses
BuildModel (off page)
updates
BuildView (off page)

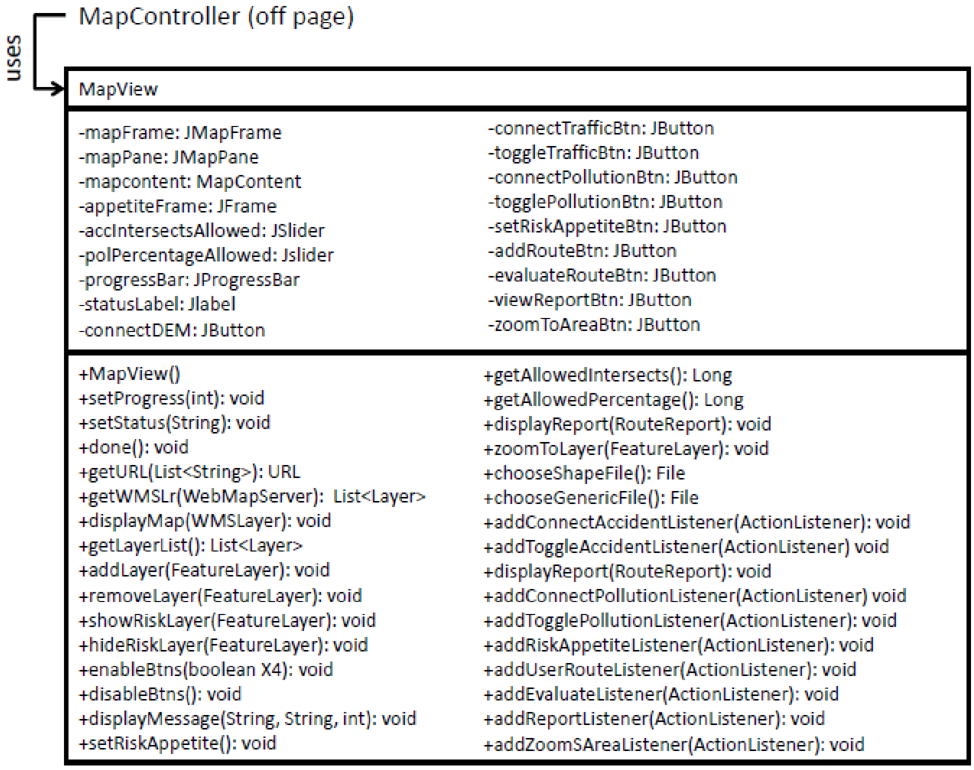

BuildController (off page)
uses
BuildModel
-PROGRESS String
-progress: int
-pcs: PropertyChangeSupport
+setProgress(int): void
+reset(): void
+addPropertyChangeListener(PropertyChangeListener): void
+getReferenceLayer(File, String): FeatureLayer
-isPolygon(SimpleFeatureSource): boolean
-isLine(SimpleFeatureSource): boolean
+buildPollutionModel(FeatureLayer, File): void
+buildTrafficModel(FeatureLayer, File, File): void
-clean(Coordinate[], int): Coordinate[]
-saveShapefile(File, SimpleFeatureType, FeatureCollection): void
uses
CSV2CoordinateImpl implements CSV2Coordinate
-line: String
+setLine(String): void
+getCoordinate(): Coordinate
-isNumeric(String): boolean
uses
CustomFeatureTypeBuilderImpl (off page)
uses
Node2FeatureImpl implements Node2Feature
-node: Node
-featureBuilder: SimpleFeatureBuilder
+setNode(Node): void
+setFeatureBuilder(SimpleFeatureBuilder): void
+getFeature(): SimpleFeature
uses
SnapImpl (off page)
uses
Snap2LineImpl implements Snap2Line
-line: LocationIndexedLine
-pt: Coordinate
-previous: Snap
+setLine(LocationIndexedLine): void
+setPoint(Coordinate): void
+setPrevious(Snap): void
+snap(): Snap
uses


BuildController (off page)
uses
BuildView
-mainFrame: JFrame
-controlPanel: JPanel
-connectPollutionGridBtn: JButton
-generatePollutionModelBtn: JButton
-connectTrafficNetworkBtn: JButton
-generateTrafficModelBtn: JButton
-progressBar: JProgressBar
-statusLabel: JLabel
+BuildView()
+setProgress(int): void
+setStatus(String): void
+done(): void
+displayGUI(): void
+enableBtns(boolean, boolean): void
+disableBtns(): void
+displayMessage(String, String, int): void
+chooseShapeFile(): File
+chooseGenericFile(): File
+setShapeFile(String): File
+addConnectPollutionGridBtnListener(ActionListener): void
+addGeneratePollutionModelBtnListener(ActionListener): void
+addConnectTrafficNetworkBtnListener(ActionListener): void
+addGenerateTrafficModelBtnListener(ActionListener): void
CSV2CoordinateImpl (off page)
uses
CustomFeatureTypeBuilderImpl implements CustomFeatureTypeBuilder
-name: String
-attributes: Map<String, Class<?>>
-crs: CoordinateReferenceSystem
-geomType: Class<?>
+setName(String): void
+setAttributes(Map<String, Class<?>>): void
+setCRS(CoordinateReferenceSystem): void
+setGeometryType(Class<?>): void
+buildFeatureType (): SimpleFeatureType
Snap2LineImpl (off page)
uses
SnapImpl implements Snap
-minDist: double
-minDistPoint: Coordinate
+setMinDist(double): void
+setMinDistPoint(Coordinate): void
+getMinDist(): double
+getMinDistPoint(): Coordinate

Analysis/Visualisation component


App
+main(String[]): void
uses
contains
BtnListener implements ActionListener (X9)
+actionPerformed(ActionEvent): void
MapController
-model: MapModel
-view: MapView
-dem: GridCoverage2D
-accidentLayer: FeatureLayer
-pollutionLayer: FeatureLayer
-userRouteLayer: FeatureLayer
-userRouteFileName: String
-report: RouteReport
-appetite: RiskAppetite
-FILE_ERROR_MSG: String
-DATA_ERROR_MSG: String
-MESSAGE_HEADING_FAIL: String
-demConnected: boolean
-accidentsConnected: boolean
-pollutionConnected: boolean
-routeLoaded: boolean
+MapController(MapModel, MapView)
+configureMapView(): void
RiskAppetiteImpl implements RiskAppetite
-maxAccidentCount: Long
-MaxPollutionPercentage: Long
+setMaxAccidentCount(Long): void
+setMaxPollutionPercentage(Long): void
+getMaxAccidentCount(): Long
+getMaxPollutionPercentage(): Long
uses
RouteReportImpl implements RouteReport
-routeFileName: String
-routeLength: Long
-slope: Double
-accidentCount: Long
-pollutionPercentage: Long
+setRouteFileName(String): void
+setRouteLength(Long): void
+setSlope(Double): void
+setAccidentCount(Long): void
+setPollutionPercentage(Long): void
+getRouteFileName(): String
+getRouteLength(): Long
+getSlope(): Double
+getAccidentCount(): Long
+getPollutionPercentage(): Long
uses
uses
MapModel (off page)
updates
MapView (off page)


MapController (off page)
uses
MapModel
-styleFactory: StyleFactory
-filterFactory: FilterFactory
-PROGRESS: String
-progress: int
-pcs: PropertyChangeSupport
-numberOfSegments: int
-counter: int
-displayCRS: CoordinateReferenceSystem
-computationCRS: CoordinateReferenceSystem
+setProgress(int): void
+reset(): void
+addPropertyChangeListener(PropertyChangeListener): void
+getServers(): List<String>
+getWMS(URL): WebMapServer
+getBackdrop(WebMapServer, Layer): WMSLayer
+getDEM(File): GridCoverage2D
+getRiskLayer(File, String): FeatureLayer
-createPolygonStyle(Color): Style
+getRouteLayer(File, String): FeatureLayer
-createLineStyle(Color): Style
+getNumIntersects(FeatureLayer, FeatureLayer): Long
+getPollutedPercentage(FeatureLayer, FeatureLayer): Long
+getRouteLength(FeatureLayer): Double
+getSlope(FeatureLayer): Double
+resetRouteStyle(FeatureLayer, RouteReport, RiskAppetite): void
uses
KML2LineImpl implements KML2Line
-routeDoc: Document
-displayCRS: CoordinateReferenceSystem
-segmentCount: int
+setDOM(Document): void
+setCRS(CoordinateReferenceSystem): void
+getLine(): DefaultFeatureCollection
+getSegmentCount(): int
-isNumeric(String): boolean


uses
MapController (off page)
MapView
-mapFrame: JMapFrame
-mapPane: JMapPane
-mapcontent: MapContent
-appetiteFrame: JFrame
-accIntersectsAllowed: JSlider
-polPercentageAllowed: Jslider
-progressBar: JProgressBar
-statusLabel: Jlabel
-connectDEM: JButton
-connectTrafficBtn: JButton
-toggleTrafficBtn: JButton
-connectPollutionBtn: JButton
-togglePollutionBtn: JButton
-setRiskAppetiteBtn: JButton
-addRouteBtn: JButton
-evaluateRouteBtn: JButton
-viewReportBtn: JButton
-zoomToAreaBtn: JButton
+MapView()
+setProgress(int): void
+setStatus(String): void
+done(): void
+getURL(List<String>): URL
+getWMSLr(WebMapServer): List<Layer>
+displayMap(WMSLayer): void
+getLayerList(): List<Layer>
+addLayer(FeatureLayer): void
+removeLayer(FeatureLayer): void
+showRiskLayer(FeatureLayer): void
+hideRiskLayer(FeatureLayer): void
+enableBtns(boolean X4): void
+disableBtns(): void
+displayMessage(String, String, int): void
+setRiskAppetite(): void
+getAllowedIntersects(): Long
+getAllowedPercentage(): Long
+displayReport(RouteReport): void
+zoomToLayer(FeatureLayer): void
+chooseShapeFile(): File
+chooseGenericFile(): File
+addConnectAccidentListener(ActionListener): void
+addToggleAccidentListener(ActionListener) void
+displayReport(RouteReport): void
+addConnectPollutionListener(ActionListener) void
+addTogglePollutionListener(ActionListener): void
+addRiskAppetiteListener(ActionListener): void
+addUserRouteListener(ActionListener): void
+addEvaluateListener(ActionListener): void
+addReportListener(ActionListener): void
+addZoomSAreaListener(ActionListener): void

Appendix 3: Implementation code fragments

Construction the traffic accident risk model

Step 1:

```
// Get the point data
BufferedReader reader = new BufferedReader(new FileReader(inputFile));
try {
       // First line of the data file is the header
       String line = reader.readLine();
       int i = 0;
       for (line = reader.readLine();line != null;line = reader.readLine()) {
             // skip blank lines
             if (line.trim().length() > 0) {
                    //Use custom made object to turn line of
                    CSV text into a coordinate
                    CSV2Coordinate csv2c = new CSV2CoordinateImpl();
                    csv2c.setLine(line);
                    Coordinate c = csv2c.getCoordinate();
                    if (c != null) {
                          tempPoints[i] = c;
                          i++;
                    }
             }
       }
} finally {
       reader.close();
}
// Method in the custom object
public Coordinate getCoordinate() {
       Coordinate c = null;
       String tokens[] = line.split("\\,");
       if (tokens.length > 6 &&
             Integer.parseInt(tokens[5]) == 1 &&
             Integer.parseInt(tokens[6]) < 3 &&
             isNumeric(tokens[3]) &&
             isNumeric(tokens[4])) {
                    double latitude = Double.parseDouble(tokens[4]);
                    double longitude = Double.parseDouble(tokens[3]);
                    c = new Coordinate(longitude, latitude);
       }
       return c;
       }

//Private helper method to check if position data is numeric before
attempting to parse (sometimes not from a gps device)
private boolean isNumeric(String str) {
       try {
             Double.parseDouble(str);
        } catch(NumberFormatException nfe) {
             return false;
        }
        return true;
    }
```

Notes:

Variable “line” is a String set by a separate mutator method to improve testability

Step 2:

```
//Get the line features of the roads and set up an iterator
FeatureSource<?,?> source = trafficReferenceNetwork.getFeatureSource();
FeatureCollection<?,?> features = source.getFeatures();
SimpleFeatureIterator lineIterator =
        (SimpleFeatureIterator) features.features();
//Place the lines in a spatial index
final SpatialIndex index = new STRtree();
try {
        while (lineIterator.hasNext()) {
                SimpleFeature lineFeature = lineIterator.next();
                Geometry geom =
                        (MultiLineString) lineFeature.getDefaultGeometry();
                if (geom != null) {
                        Envelope env = geom.getEnvelopeInternal();
                                if (!env.isNull()) {
                                        index
                                        .insert(env, new LocationIndexedLine(geom));
                                }
                }
        }
} finally {
        lineIterator.close();
}

//The maximum distance that a line can be from a point to be a candidate for
snapping is defined as 1% of the total width of the feature bounds
ReferencedEnvelope bounds = features.getBounds();
final double MAX_SEARCH_DISTANCE = bounds.getSpan(0) / 100.0;
int pointsProcessed = 0;
while (pointsProcessed < NUM_POINTS) {

        //Get point and create search envelope
         Coordinate pt = points[pointsProcessed++];
         Envelope search = new Envelope(pt);
         search.expandBy(MAX_SEARCH_DISTANCE);

         //Query the spatial index for objects within the search envelope.
         List<LocationIndexedLine> lines = index.query(search);

         //Initialise the minimum distance found to the maximum acceptable
         distance plus a little bit
         double minDist = MAX_SEARCH_DISTANCE + 1.0e-6;
         Coordinate minDistPoint = null;
        for (LocationIndexedLine line : lines) {
                //Use custom made objects to snap the point to a line
                Snap firstSnap = new SnapImpl();
                firstSnap.setMinDist(minDist);
                firstSnap.setMinDistPoint(minDistPoint);
                Snap2Line s2l = new Snap2LineImpl();
                s2l.setLine(line);
                s2l.setPoint(pt);
                s2l.setPrevious(firstSnap);
                Snap returnedSnap = s2l.snap();
                minDist = returnedSnap.getMinDist();
                minDistPoint = returnedSnap.getMinDistPoint();
        }
```

```
    if (minDistPoint != null) {
        Point point = geometryFactory.createPoint(minDistPoint);
        featureBuilder.add(point);
        SimpleFeature feature = featureBuilder.buildFeature(null);
        pointCollection.add(feature);
    }
}
// Method in the custom object
public Snap snap() {
    double minDist = previous.getMinDist();
    Coordinate minDistPoint = previous.getMinDistPoint();
    LinearLocation here = line.project(pt);
    Coordinate point = line.extractPoint(here);
    double dist = point.distance(pt);
    if (dist < minDist) {
        minDist = dist;
        minDistPoint = point;
    }
    Snap newSnap = new SnapImpl();
    newSnap.setMinDist(minDist);
    newSnap.setMinDistPoint(minDistPoint);
    return newSnap;
}
```

Notes:

Separate mutator methods are provided to set the candidate point (pt), the line to evaluate against (line) and the previously seen best candidate (previous) in order to improve testability.

Steps 3 & 4:

```
MathTransform transform = CRS.findMathTransform(crs, wmsCRS, true);

//Set up an empty feature collection to hold buffer features once created
DefaultFeatureCollection bufferCollection = new DefaultFeatureCollection();

SimpleFeatureIterator pointIterator = pointCollection.features();
try {
    while (pointIterator.hasNext()) {
        SimpleFeature pointFeature = pointIterator.next();
        Point inputPoint = (Point) pointFeature.getDefaultGeometry();
        Geometry transformedPoint =
            JTS.transform(inputPoint, transform);
        Geometry buffer = transformedPoint.buffer(25.0);
        bufferBuilder.add(buffer);
        SimpleFeature feature = bufferBuilder.buildFeature(null);
        bufferCollection.add(feature);
     }
} finally {
    pointIterator.close();
}
```

Notes:

The functionality required to build feature types and also to save a shapefile consist of calls to standard Geotools classes and is not given here.

Construction the air-pollution risk model

Step 1:

```
//Set up an empty feature collection to hold point features once created
DefaultFeatureCollection pointCollection = new DefaultFeatureCollection();

//Get the data feed and create a DOM
InputStream inputStream = null;
String lAir = "http://api.erg.kcl.ac.uk/AirQuality/Hourly/MonitoringIndex/
GroupName=London";
URL url = new URL(lAir);
URLConnection conn = url.openConnection();
conn.connect();
inputStream = conn.getInputStream();
Document doc = DOMUtils.readXml(inputStream);
inputStream.close();
NodeList nList = doc.getElementsByTagName("Site");

//Iterate through the nodes
for (int i=0; i<nList.getLength(); i++) {
        Node nNode = nList.item(i);

        //Use custom object to turn nodes into point features
        Node2Feature n2f = new Node2FeatureImpl();
        n2f.setNode(nNode);
        n2f.setFeatureBuilder(featureBuilder);
        SimpleFeature feature = n2f.getFeature();

        //Add the point to the collection
        pointCollection.add(feature);
}

// Method in the custom object
public SimpleFeature getFeature() {
        double lat = Double.parseDouble(node.getAttributes()
                .getNamedItem("Latitude").getNodeValue());
        double lon = Double.parseDouble(node.getAttributes()
                .getNamedItem("Longitude").getNodeValue());
        String sitename = node.getAttributes()
                .getNamedItem("SiteName").getNodeValue();
        NodeList sList = node.getChildNodes();
        double total = 0.0;
        int validMeasures = 0;
        for (int j=0; j<sList.getLength(); j++) {
                Node sNode = sList.item(j);
                double reading = Double.parseDouble(sNode.getAttributes()
                        .getNamedItem("AirQualityIndex").getNodeValue());
                if (reading > 0) {
                        total = total + reading;
                        validMeasures = validMeasures + 1;
                }
        }
        double val = total/validMeasures;
        Coordinate c = new Coordinate(lat, lon);
        GeometryFactory geometryFactory = JTSFactoryFinder
                .getGeometryFactory(null);
        Point point = geometryFactory.createPoint(c);
        featureBuilder.add(sitename);
        featureBuilder.add(val);
        featureBuilder.add(point);
        SimpleFeature feature = featureBuilder.buildFeature(null);
        return feature;
}
```

Notes:

Separate mutator methods are provided to set the DOM node and the feature builder object in order to improve testability.

Step 2a:

```
//Set up an empty temporary feature collection to hold new polygon features 
once created
DefaultFeatureCollection tempPolyCollection = 
        new DefaultFeatureCollection();

//Get the feature collection of input polygons and make an iterator
FeatureCollection<?,?> inputPolyCollection = pollutionReferenceGrid
        .getFeatureSource().getFeatures();
SimpleFeatureIterator inputPolyIterator = 
        (SimpleFeatureIterator) inputPolyCollection.features();

try {
        while(inputPolyIterator.hasNext()) {

                //Prepare the polygon feature
                SimpleFeature polyFeature = inputPolyIterator.next();
                MultiPolygon poly = 
                        (MultiPolygon) polyFeature.getDefaultGeometry();
                Point interpoint = poly.getCentroid();
                Geometry transformedPoint = 
                        JTS.transform(interpoint, mercatorToGeographic);
                gc.setStartingPosition(JTS
                                .toDirectPosition(transformedPoint
                                .getCoordinate(), crs));

                //Use custom made object to calculate interpolated values
                IDWinterpolation idw = new IDWinterpolationImpl();
                idw.setCalculator(gc);
                idw.setPointCollection(pointCollection);
                double interpolation = idw.interpolate();

                //Write interpolated values back to the feature attribute
                polyFeature.setAttribute("value", interpolation);
                tempPolyCollection.add(polyFeature);
           }
} finally {
        inputPolyIterator.close();
}
```

Step 2b:

```
public Double interpolate() throws TransformException {
    double numerator = 0.0;
    double denominator = 0.0;
    SimpleFeatureIterator pointIterator =
        (SimpleFeatureIterator) pointCollection.features();
    try {
        while(pointIterator.hasNext()) {
            SimpleFeature pointFeature = pointIterator.next();
                double pointval =
                    (Double) pointFeature
                        .getAttribute("pollution");
                if (pointval > 0) {
                    Point valuepoint =
                        (Point) pointFeature
                            .getDefaultGeometry();
                    gc.setDestinationPosition(JTS
                        .toDirectPosition(valuepoint
                        .getCoordinate(), crs));
                double distance = gc.getOrthodromicDistance();
                    if (distance < 5000) {
                        numerator = numerator +
                            (pointval/Math
                                .pow(distance, 2));
                        denominator = denominator +
                            (1/Math.pow(distance, 2));
                    }
                }
        }
    } finally {
        pointIterator.close();
    }
    double interpolation = 0.0;
    if (denominator > 0) {
        interpolation = numerator/denominator;
    }
    return interpolation;
}
```

Notes:

Separate mutator methods are provided to set the geodetic calculator object (gc) and the point collection in order to improve testability.

Step 3:

```
//Calculate cut-off point for inclusion in model
FilterFactory2 ff = CommonFactoryFinder.getFilterFactory2();
Function sum = ff.function("Collection_Max", ff.property("value"));
Object value = sum.evaluate(tempPolyCollection);
double max = (double) value;
double topSlice = max - (max/3);
```

```
//Set up an output collection and an iterator
DefaultFeatureCollection outputPolyCollection = new
DefaultFeatureCollection();
SimpleFeatureIterator outputPolyIterator =
	(SimpleFeatureIterator) tempPolyCollection.features();
try {
	while(outputPolyIterator.hasNext()) {
		SimpleFeature newPolyFeature = outputPolyIterator.next();

		 //Evaluate features and include if appropriate
		  if ((Double)newPolyFeature.getAttribute("value") > topSlice) {
			   outputPolyCollection.add(newPolyFeature);
		  }
	}
} finally {
	outputPolyIterator.close();
}
```

Notes:

The functionality required to build feature types and also to save a shapefile consist of calls to standard Geotools classes and is not given here.

Processing of the user route

```
public DefaultFeatureCollection getLine() throws IOException {
	CoordinateReferenceSystem computationCRS = null;
	try {
		computationCRS = CRS.decode("EPSG:4326");
	} catch (NoSuchAuthorityCodeException e) {
		// Only caused by coding errors
		e.printStackTrace();
	} catch (FactoryException e) {
		// Only caused by coding errors
		e.printStackTrace();
	}
	DefaultFeatureCollection lineCollection =
		new DefaultFeatureCollection();

	routeDoc.getDocumentElement().normalize();
	NodeList nList = routeDoc.getElementsByTagName("coordinates");
	if (nList.getLength() < 1) {
		throw new IOException();
	}
	Node cNode = nList.item(0);
	String raw = cNode.getTextContent();
	String[] masterArray = raw.split(" ");
	Coordinate[] tempcoords = new Coordinate[masterArray.length];
	int index = 0;
```

```
    for (String s : masterArray) {
        String[] subArray = s.split(",");
        if (subArray.length > 2 &&
            isNumeric(subArray[1]) &&
            isNumeric(subArray[0])) {
            Coordinate c =
                new Coordinate(Double.parseDouble(subArray[1]),
                Double.parseDouble(subArray[0]));
            tempcoords[index] = c;
            index++;
        }
    }
    Coordinate[] coords = new Coordinate[index];
    for (int n=0; n<index; n++) {
        coords[n] = tempcoords[n];
    }

    for (int i=0; i<coords.length-1; i++){
        Coordinate[] outCoords = {coords[i], coords[i+1]};
        LineString line = geometryFactory.createLineString(outCoords);
        MathTransform transform = null;
        try {
            transform =
            CRS.findMathTransform(computationCRS, displayCRS, true);
        } catch (FactoryException e) {
            // Only caused by coding errors
            e.printStackTrace();
            }
        Geometry transformedLine = null;
        try {
            transformedLine = JTS.transform(line, transform);
        } catch (MismatchedDimensionException e) {
            // Only caused by coding errors
            e.printStackTrace();
        } catch (TransformException e) {
            // Only caused by coding errors
            e.printStackTrace();
        }

        featureBuilder.add(transformedLine);
        SimpleFeature feature = featureBuilder.buildFeature(null);
        lineCollection.add(feature);
    }

    return lineCollection;
    }
```

Notes:

Separate mutator methods are provided to set the DOM (routedoc) and the CRS (displayCRS) in order to improve testability.

Calculating route length

```
public double getRouteLen(FeatureLayer userRouteLayer) throws
        NoSuchAuthorityCodeException, FactoryException, IOException,
        MismatchedDimensionException, TransformException {

        double val = 0.0;
        FeatureCollection<?, ?> lineCollection =
                userRouteLayer.getFeatureSource().getFeatures();
        MathTransform transform =
                CRS.findMathTransform(displayCRS, computationCRS, true);
        GeodeticCalculator gc = new GeodeticCalculator(computationCRS);

        SimpleFeatureIterator iterator =
                (SimpleFeatureIterator) lineCollection.features();

        try {
                while(iterator.hasNext()){
                        SimpleFeature feature = iterator.next();
                        MultiLineString geom =
                                (MultiLineString) feature.getDefaultGeometry();
                        int n = geom.getNumGeometries();
                        LineString lines[] = new LineString[n];
                        for ( int i = 0; i < n; i++ ) {
                                lines[i] = (LineString) geom.getGeometryN(i);
                                Point startP = lines[i].getStartPoint();
                                Point endP = lines[i].getEndPoint();
                                Geometry startPTransformed =
                                        JTS.transform(startP, transform);
                                Geometry endPTransformed =
                                        JTS.transform(endP, transform);
                                Coordinate start =
                                        startPTransformed.getCoordinate();
                                Coordinate end = endPTransformed.getCoordinate();
                                gc.setStartingPosition(JTS
                                        .toDirectPosition(start, computationCRS));
                                gc.setDestinationPosition(JTS
                                        .toDirectPosition(end, computationCRS));

                                double distance = gc.getOrthodromicDistance();
                                val = val + distance;
                        }
                 }
        } finally {
                iterator.close();
        }
        return val;
}
```

Calculating slope gradient

```
public double getSlope(FeatureLayer userRouteLayer, GridCoverage2D dem)
    throws IOException, MismatchedDimensionException,
        NoSuchAuthorityCodeException, FactoryException,
        TransformException {

    MathTransform transform =
        CRS.findMathTransform(displayCRS, computationCRS, true);
    FeatureCollection<?, ?> lineCollection =
        userRouteLayer.getFeatureSource().getFeatures();
    SimpleFeatureIterator lineIterator =
        (SimpleFeatureIterator) lineCollection.features();
    SimpleFeature lineFeatureStart = lineIterator.next();
    MultiLineString mLineS =
        (MultiLineString) lineFeatureStart.getDefaultGeometry();
    LineString firstLine = (LineString) mLineS.getGeometryN(0);

    LineString lastLine = null;
    SimpleFeature lineFeatureEnd = null;
    try {
        while(lineIterator.hasNext()) {
            lineFeatureEnd = lineIterator.next();
        }
    } finally {
        lineIterator.close();
    }

    MultiLineString mLineE =
        (MultiLineString) lineFeatureEnd.getDefaultGeometry();
    lastLine = (LineString) mLineE.getGeometryN(0);

    Point startP = firstLine.getStartPoint();
    Point endP = lastLine.getStartPoint();
    Geometry startPTransformed = JTS.transform(startP, transform);
    Geometry endPTransformed = JTS.transform(endP, transform);
    Coordinate startC = startPTransformed.getCoordinate();
    Coordinate endC = endPTransformed.getCoordinate();

    float[] startHeight =
        (float[]) (dem
            .evaluate(JTS.toDirectPosition(startC, computationCRS)));
    float[] endHeight =
        (float[]) (dem
            .evaluate(JTS.toDirectPosition(endC, computationCRS)));

    double val =
    ((startHeight[0]-endHeight[0])/getRouteLen(userRouteLayer)) * 100;
    return val;

    }
```

Calculating pollution percentage

```java
public int getPollutedPercentage(FeatureLayer userRouteLayer,
    FeatureLayer riskLayer) throws NoSuchAuthorityCodeException,
    FactoryException, IOException, MismatchedDimensionException,
    TransformException {

    int val = 0;
    FeatureCollection<?, ?> lineCollection =
        userRouteLayer.getFeatureSource().getFeatures();
    FeatureCollection<?, ?> polyCollection =
        riskLayer.getFeatureSource().getFeatures();
    SimpleFeatureIterator lineIterator =
        (SimpleFeatureIterator) lineCollection.features();
    double polDist = 0.0;
    try {
    while( lineIterator.hasNext() ){
        SimpleFeatureIterator polyIterator =
        (SimpleFeatureIterator) polyCollection.features();
         SimpleFeature lineFeature = lineIterator.next();
         MultiLineString lines =
        (MultiLineString) lineFeature.getDefaultGeometry();
        try {
        while (polyIterator.hasNext()) {
            SimpleFeature polyFeature = polyIterator.next();
            MultiPolygon polys =
            (MultiPolygon) polyFeature.getDefaultGeometry();
            int np = polys.getNumGeometries();
            Polygon polyArray[] = new Polygon[np];
            for ( int j = 0; j < np; j++ ) {
                polyArray[j] = (Polygon) polys.getGeometryN(j);
                if (lines.intersects(polyArray[j])) {
                    Geometry pollutedPart =
                    lines.intersection(polyArray[j]);
                    int nl = pollutedPart.getNumGeometries();
                    LineString lineArray[] =
                    new LineString[nl];
                    for (int k=0; k<nl; k++) {
                        lineArray[k] =
                        (LineString) pollutedPart
                            .getGeometryN(k);
                        polDist = polDist +
                        getLineStringLength(lineArray[k]);
                    }
                }
            }
        }
        } finally {
            polyIterator.close();
       }
    }
    } finally {
        lineIterator.close();
    }

    double totDist = getRouteLen(userRouteLayer);
    val = (int) ((polDist/totDist) *100);
    return val;
  }
```

Calculating accident count

```
public int getNumIntersects(FeatureLayer userRouteLayer,
    FeatureLayer riskLayer) throws IOException {

    int val = 0;
    FeatureCollection<?, ?> lineCollection =
    userRouteLayer.getFeatureSource().getFeatures();
    FeatureCollection<?, ?> polyCollection =
    riskLayer.getFeatureSource().getFeatures();

    Set<String> polyIds = new HashSet<String>();

    SimpleFeatureIterator lineIterator =
        (SimpleFeatureIterator) lineCollection.features();

    try {
    while( lineIterator.hasNext() ){
        SimpleFeature lineFeature = lineIterator.next();
        MultiLineString lines =
        (MultiLineString) lineFeature.getDefaultGeometry();
        SimpleFeatureIterator polyIterator =
        (SimpleFeatureIterator) polyCollection.features();
         try {
         while (polyIterator.hasNext()) {
            SimpleFeature polyFeature = polyIterator.next();
            MultiPolygon polys =
                (MultiPolygon) polyFeature.getDefaultGeometry();
            int np = polys.getNumGeometries();
            Polygon polyArray[] = new Polygon[np];

            for ( int j = 0; j < np; j++ ) {
                polyArray[j] = (Polygon) polys.getGeometryN(j);
                if (lines.intersects(polyArray[j])) {
                    polyIds.add(polyFeature.getID());
                }
            }
        }
        } finally {
            polyIterator.close();
        }
    }
    } finally {
        lineIterator.close();
    }
    val = polyIds.size();
    return val;
}
```

Appendix 4: Validation code

Obtaining the feed data using Java

```
public static void main(String[] args) {
        InputStream inputStream = null;
        OutputStream outputStream = null;
        String lAir =
                "http://api.erg.kcl.ac.uk/AirQuality/Hourly/MonitoringIndex/
                        GroupName=London";
        try {
                URL url = new URL(lAir);
                URLConnection conn = url.openConnection();
                conn.connect();
                inputStream = conn.getInputStream();
                outputStream =
                        new FileOutputStream(
                        new File("C:\\mscprojek\\feeddata.xml"));
                int read = 0;
                byte[] bytes = new byte[1024];
                while ((read = inputStream.read(bytes)) != -1) {
                        outputStream.write(bytes, 0, read);
                }
                outputStream.close();

                } catch (MalformedURLException e) {
                        e.printStackTrace();
                } catch (IOException e) {
                        e.printStackTrace();
                }
        }
}
```

Creating a CSV file using Python

```
from lxml import etree
import csv

csvfile = open(r'c:\mscprojek\feeddata.csv', 'wb')
mywriter = csv.writer(csvfile, delimiter=',')
mywriter.writerow(['SiteName','Latitude', 'Longitude', 'AvgPolution'])

def find_elements(tree):
    my_elements = tree.xpath("/HourlyAirQualityIndex")
    return my_elements

parser = etree.XMLParser(load_dtd=False)
input_file = r"C:\mscprojek\feeddata.xml"
tree = etree.parse(input_file, parser)
my_elements = find_elements(tree)

for e in my_elements:
    la_list = e.findall("LocalAuthority")
```

```
for e in la_list:
    site_list = e.findall("Site")

    for e in site_list:
        site = e.get("SiteName")
        lat = e.get("Latitude")
        lon =  e.get("Longitude")

        species_list = e.findall('Species')

        total = 0
        for s in species_list:
            total = total + float(s.get("AirQualityIndex"))

        avg = total/len(species_list)
        mywriter.writerow([site, lat, lon, avg])

csvfile.close()
```

Appendix 6: POM file dependencies node

```xml
<dependencies>
        <dependency>
                <groupId>junit</groupId>
                <artifactId>junit</artifactId>
                <version>4.11</version>
                <scope>test</scope>
        </dependency>
        <dependency>
                <groupId>org.geotools</groupId>
                <artifactId>gt-shapefile</artifactId>
                <version>${geotools.version}</version>
        </dependency>
        <dependency>
                <groupId>org.geotools</groupId>
                <artifactId>gt-swing</artifactId>
                <version>${geotools.version}</version>
        </dependency>
        <dependency>
                <groupId>org.geotools</groupId>
                <artifactId>gt-epsg-hsql</artifactId>
                <version>${geotools.version}</version>
        </dependency>
        <dependency>
                <groupId>org.geotools</groupId>
                <artifactId>gt-geotiff</artifactId>
                <version>${geotools.version}</version>
        </dependency>
        <dependency>
                <groupId>org.geotools</groupId>
                <artifactId>gt-image</artifactId>
                <version>${geotools.version}</version>
        </dependency>
        <dependency>
                <groupId>org.geotools</groupId>
                <artifactId>gt-wms</artifactId>
                <version>${geotools.version}</version>
        </dependency>
                <dependency>
                <groupId>org.apache.cxf</groupId>
                <artifactId>cxf-api</artifactId>
                <version>2.7.1</version>
        </dependency>
 </dependencies>
```