# Design

## Introduction

This is the design section of the project, where the functionality, classes and processes of the program are described as they were designed.

## IPSO chart

This chart outlines the processes in the program at a basic level, in terms of input/output, processing, and storage. A full detail of these processes is described below.

|  |  |  |  |
| --- | --- | --- | --- |
| **Input** | **Processing** | **Storage** | **Output** |
| **Departure bus stop**  **Arrival bus stop**  **Time and date**  **Arrive before or depart after time option**  **Bus Change Risk\*** | **Retrieve appropriate timetable from the database**  **Calculating the shortest journey time**  **Drawing interactive map**  **Create a printable document** | **Database tables:**  ***Bus Stops***  ***Buses***  ***Links***  ***Timetable***  ***Day Categories***  ***Special Days*** | **Digital document**  **Printed document**  **Interactive map** |

\*Bus Change Risk is the time gap in minutes between bus changes.

## System Flow Diagram

Here is a flow diagram representing the flow of processes in the new proposed system.

(flowchart missing)

At start up

Load Core Data from now on it includes bus stops, links, buses, day categories and special days

Load timetable according to the present date

If loading of the timetable was successful then proceed else display an error that data could not be loaded

Draw bus stops and links on map, and add bus stop names to the drop down menu

When user selects bus stops from the drop down menus or map, the bus stops appropriate bus stops highlight on map.

When the calculate button clicked.

## User Interface Design

Here are some proposed sketches of the layout of the interface.

Design 1

(scan missing)

Design 2

(scan missing)

The design proposals share 3 common interface elements (I will describe these in more detail later on in the Design Section):

1. Search panel – here the user chooses the start of journey bus stop and the destination bus stop, time and date of the journey and bus change risk.
2. Map – the user can also choose the bus stops as well and see the results of the calculated bus journey by highlighted path. The map should be interactive and highlight the bus stops as the cursor is moved over then. The map will also have a zoom in and out feature so that big networks can fit in the space provided.
3. Feedback table – the user will be able to see and compare calculated bus journeys here. Basic information concerning the departure time and the arrival time at the destination bus stop and the number of changes. The user will be able to choose to display more information and print or save it.

After a discussion with the user, we decided Design 2 would be more applicable and visually easier to use as it would display all helpful information at once.

Design 2 detail

(1) Journey Search Panel

(2) Interactive map

(3)  
Feedback form stack panel

(4) About button

(5) Help button

(6) Print button

(7) Clear button

User Interaction:

1. Journey Search Panel – The user specifies the start and end of journey bus stops, time and date, and selects whether to arrive before or after the specified time. The user can select bus stops from a drop down menu, avoids spelling error of the user.
2. Interactive map – an abstract map of the network is displayed from which the user can also select the start and end bus stops by selecting from the right click menu. Bus stops highlight when mouse moves over them. The bus stops in the journey will change colour to show the path of the journey. There will be 4 different colours: start bus stop, end bus stop, in journey bus stop, bus change bus stop.
3. Feedback Forms – the calculated journey will be displayed in a feedback form with basic information: departure time, arrival time, total duration of journey for comparison, and the number of changes. Available will be two buttons one for showing more information and one for printing. When the cursor is moved over a feedback-form, the journey will be highlighted on the map.
4. About button – shows basic information about the application, the producer’s name and the name of the program, etc.
5. Shows up a new window with the user manual.
6. Prints ALL the journeys stored in the feedback form stack panel
7. Clear everything from the feedback form stack panel

## Algorithms

To calculate the quickest path the program will use an alteration of the well known Dijkstra’s Algorithm looking for the shortest distance between two nodes with fixed weighted edges. I will describe the original algorithms first before proceeding to the algorithm that will be used by the program to explain the subtle differences.

Original Dijkstra’s Algorithm:

Each node has to store the previous best link to that node. The first node is a special case; since it is a starting point it does not have the best link. The idea of the algorithm is to continuously calculate the shortest distance beginning from a source node while excluding the longer distances when making an update.

Here is the pseudo code for the algorithm:

**function Dijkstra(Graph, source):**

*//Initialisation*

*// dist[node] is the distance from source to node*  
**dist[source] ← 0***// dist[source] is the distance from source to source*

*// prev[node] is the previous node that linked with the node*  
**prev[source] ← undefined***// prev[source] is the previous node that linked with source*  
  
*// For the rest of the nodes in the Graph*  
 **for each vertex v in Graph:  
 if v ≠ source  
 dist[v] ← infinity** *// The distance from source is set to be very large*

**prev[v] ← undefined** *// The previous node is set to be undefined* **end if**

**add v to Unvisited**   
*// Add all nodes initially to Unvisited (nodes)*

**end for**

*//Calculating*

**while Unvisited is not empty:** **u ← vertex in Unvisited with min dist[u]** *// Source node in first case*

**remove u from Unvisited**

**for each neighbour v of u:** *// where v is still in Unvisited* **alt ← dist[u] + length(u, v)**   
 *// length is the fixed weight between the nodes  
 // alt is the new length calculated* **if alt < dist[v]:** *// i.e. a shorter path to v has been found* **dist[v] ← alt  
 prev[v] ← u   
 end if  
 end for  
 end while**

**return dist[], prev[]**

**end function**

Based on this algorithm there will be two algorithms that calculate the shortest path between any two bus stops. The one of the alterations has to take into account that the weight of the edges is timetable dependent, meaning that the weight depends on the time and date and might vary throughout the day; also, ‘weight’ is not simply a number but a time of departure from a node to the arrival time to the next node. Another alteration has to take into account the fact that there could be several links made by different buses between same two nodes. The minimum time taken during bus changeover also has to be taken into consideration.

For this special classes have to be created and made so that the timetable of each bus stop can be easily accessed. The classes are described above.

The pseudo code of the algorithm (alterations are in **green**):

**function Dijkstra(Graph, source):**

*//Initialisation*  **arrTime[source] ← 0 //  
depTime[source] ← 0 //**  
**prevNode[source] ← undefined // previous node  
prevBus[source] ← undefined // previous bus  
prevArr[source] ← undefined //**

*// for the rest of the nodes in the Graph*  
 **for each vertex v in Graph:  
 if v ≠ source  
 dist[v] ← infinity** *// The distance from source is set to be very large*

**prev[v] ← undefined** *// The previous node is set to be undefined* **end if**

**add v to Unvisited**   
*// Add all nodes initially to Unvisited (nodes)*

**end for**

*//Calculating*

**while Unvisited is not empty:** **u ← vertex in Unvisited with min dist[u]** *// Source node in first case*

**remove u from Unvisited**

**for each neighbour v of u:** *// where v is still in Unvisited* **alt ← dist[u] + length(u, v)**   
 *// length is the fixed weight between the nodes  
 // alt is the new length calculated* **if alt < dist[v]:** *// i.e. a shorter path to v has been found* **dist[v] ← alt  
 prev[v] ← u   
 end if  
 end for  
 end while**

**return dist[], prev[]**

**end function**

Database to access from:

Data Dictionary

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Field | Data Type | Length |  |  |  |  |

More detailed system flow chart

Modular Design

Form/Navigation Based

Main: calculate journey and show result on map

Code based

Classes and inheritance

Definition of data requirements (Design Data Dictionary) – copy and paste from analysis

Definition of record structure:

Validation required

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Validation Check | Description | Suitable Applicable Field | Valid data | Erroneous Data |
| Datatype | Checks if the name of the bus stop is correct – this can be implemented using a drop down menu to ensure that only existing bus stops are inserted. |  | Cambridge City Centre | Cambrige citi center |
| Length | The date needs to be valid, i.e. | Postcode | E4 7HJ | EEE668 JUY |
| Range | Checks that the data is within a set range, being greater than, less than, between, etc. For example the age of someone will be greater than 0, the number of days in a month would be greater or equal to 28 and less than 31. | Number of kidneys | 2 | 45 |
| Presence | This is when you need to make inputting data compulsory, the user isn't allowed to miss a field out. For example, they must state whether they agree or disagree with a legal statement, they must enter a postcode | Phone number | 07070707070 | (blank) |
| Lookup/List | This is when you only allow set values into a field. For example a Title would only allow for "Mr.", "Ms.", "Miss.", "Mrs.", "Dr.". Anything else would be erroneous. You can implement this normally by using a drop down list | Gender | Female | Jackal |

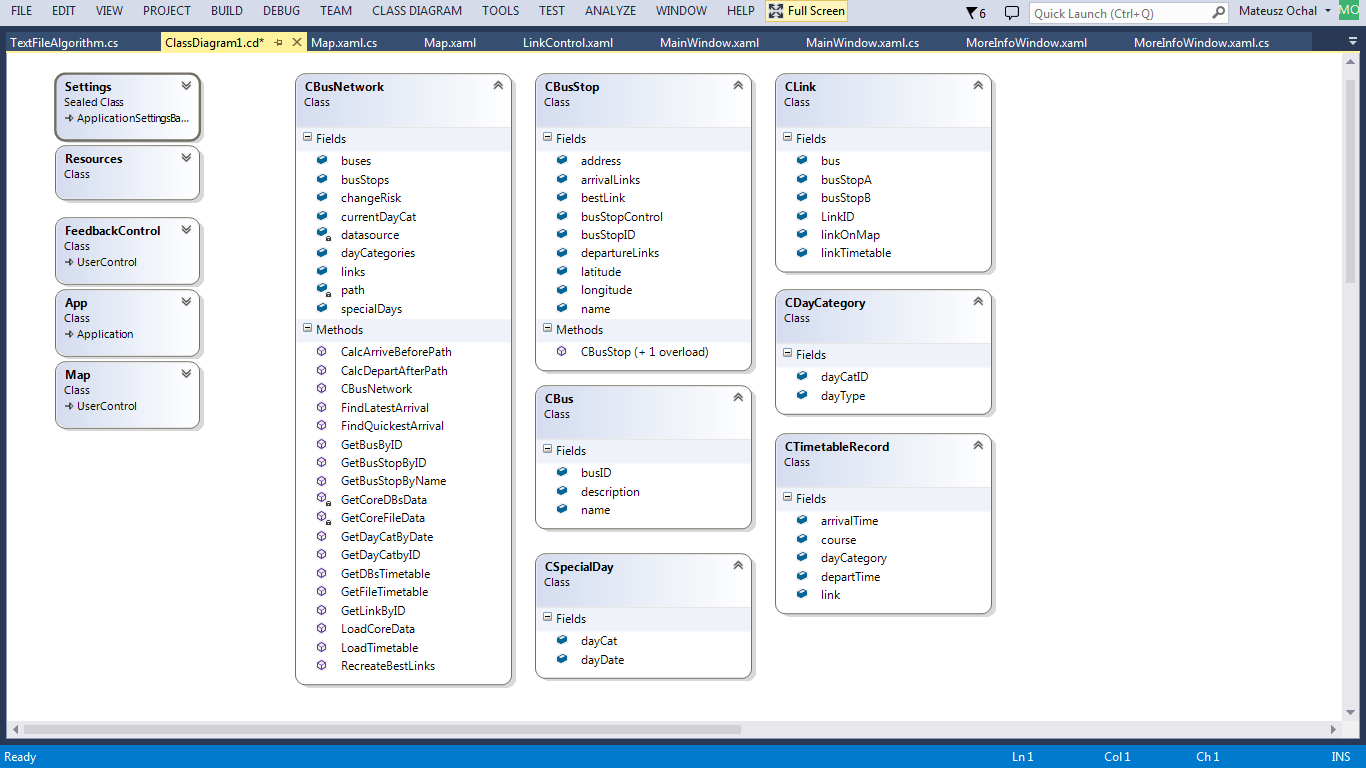
Testing the depart after algorithm

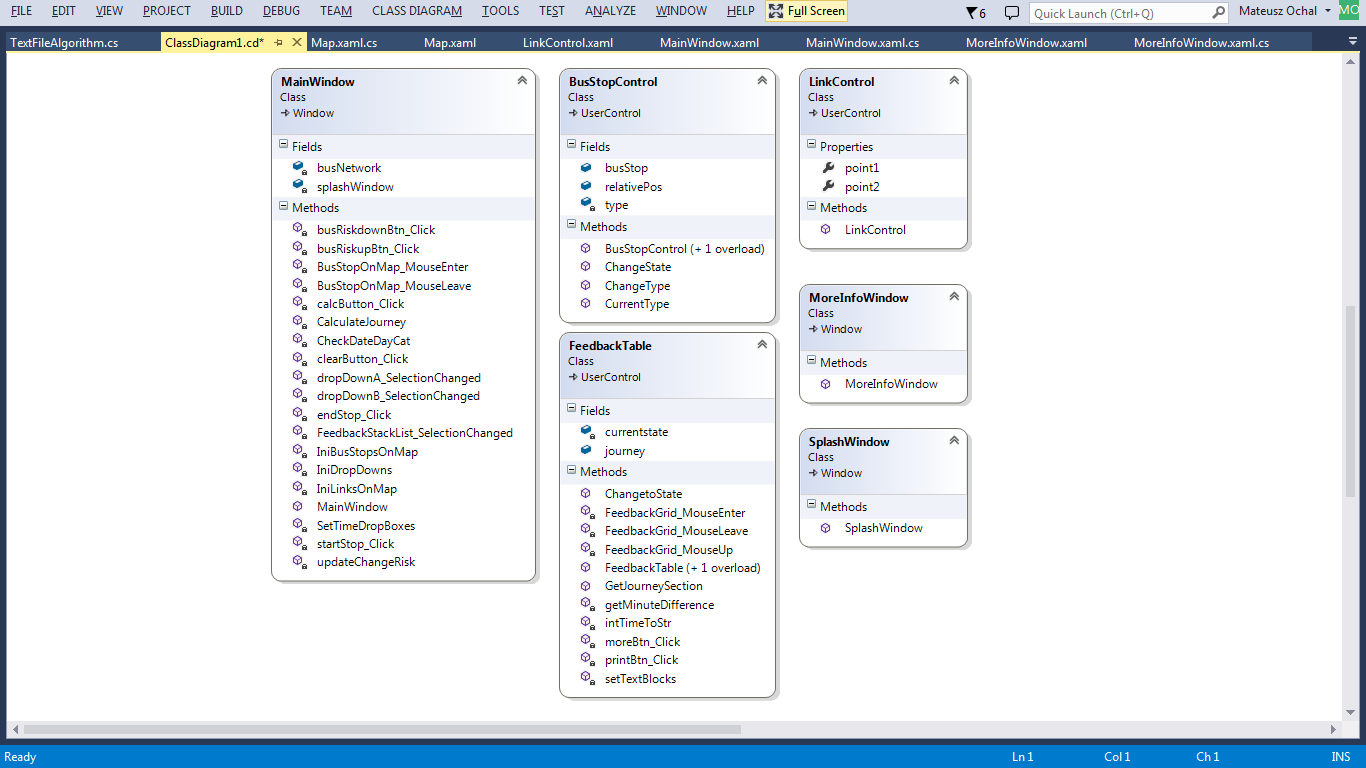
For this network will be tested on smaller bus networks

* Small Network:

|  |  |  |  |
| --- | --- | --- | --- |
| Test Number | Description | Data type | Expected result |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

Class Diagrams





Algorithms:

The algorithms used are based on the Dijkstra’s algorithm for finding the shortest path between two nodes with fixed weight on each edge.

Depart After Algorithm

Variables:

* UnvisitedStops – a list with all the bus stops or nodes in the system to begin with
* CurrentStop – the bus stop for an instance in time
* DepartureTime – the time when the bus leaves a bus stop
* ArrivalTime – the time when the bus arrives at the neighbouring bus stop

Psoudocode:

currentBusStop = FindQuickestArrival(unvisitedBusStops);

unvisitedBusStops.Remove(currentBusStop);

/// This is the main loop. It works by finding the "best link record" like in the loop above.

/// Going through each bus stop in turn, until the "current bus stop" is the "end bus stop"

/// OR

///

while ((currentBusStop != endBusStop) && (unvisitedBusStops.Count > 0))

{

foreach (CLink link in currentBusStop.departureLinks){

//find the ones with departure time straight after the given time in the

foreach (CTimetableRecord record in link.linkTimetable){

if ((((record.course == currentBusStop.bestLinkRecord.course) & (record.link.bus == currentBusStop.bestLinkRecord.link.bus)) || (record.departTime >= currentBusStop.bestLinkRecord.arrivalTime + changeRisk)) && (record.arrivalTime < record.link.busStopB.bestLinkRecord.arrivalTime))

{

if (unvisitedBusStops.Contains(record.link.busStopB))

record.link.busStopB.bestLinkRecord = record;

}

}

}

currentBusStop = FindQuickestArrival(unvisitedBusStops); //choose the busstop with the quickest arrival time out of the unvisited bus stops

unvisitedBusStops.Remove(currentBusStop);

}