Exercise 2 – Navigation System (Improved Design)

In this exercise, you have to extend the navigation system that you implemented in the last exercise.

In the first part of the exercise, you have to make some design decisions. **Note: Only handwritten answers and diagrams will be accepted.** For the implementation, you should proceed as follows:

- Create a new project in Eclipse
- Copy the code from the last exercise to your new project (only content of myCode folder
- Create the new classes using Together
- Implement and test the classes one by one on Eclipse (provide stubs as required)

Marking:

2.1	• Polymorphic route design	3
2.2	• Polymorphic route implementation + Algorithms	3
2.3	• Persistance	3

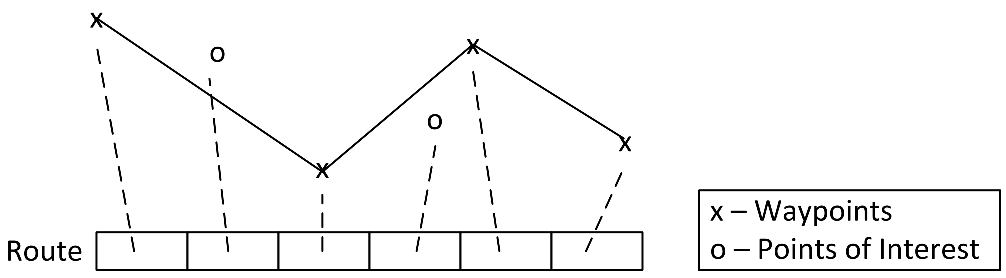
Source, Internet

Please note that the allocation of time slots will change for the second mandatory lab. MAKE SURE TO CHECK YOUR SLOT TWO DAYS BEFORE YOUR REVIEW DATE. NOT BEING THERE ON TIME WILL RESULT IN A ZERO POINT GRADING.

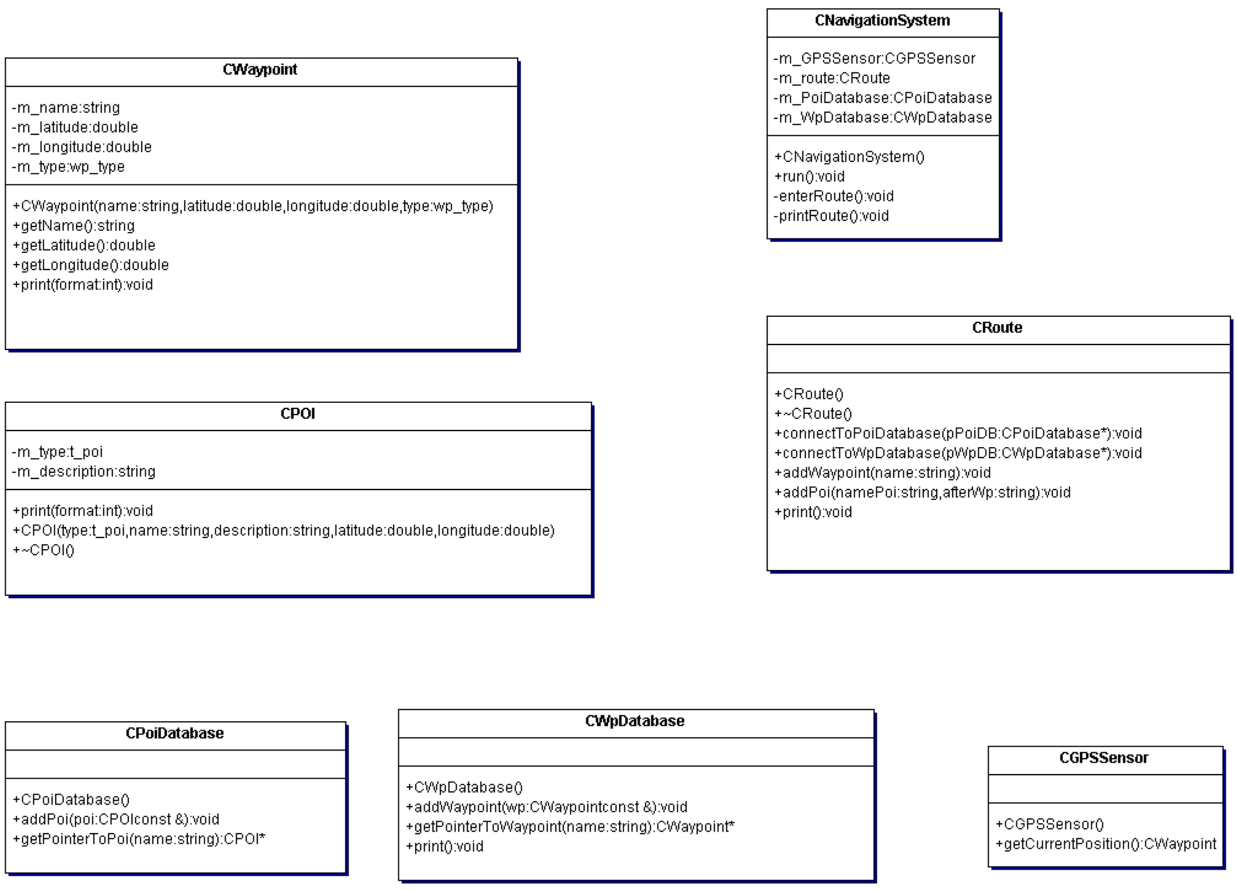
1.1 Polymorphic route design

Until now we have designed a rather inelegant implementation of the route, providing different containers for the waypoints and the points of interest of the route. A better implementation would be the following polymorphic design:

- All waypoints and POI's are stored in separate databases
- The route itself is stored in a single common container, which contains both the waypoints and the POI's in the order the driver will reach them (sketch below).



Add all class relations to the diagram below. Do not forget the multiplicities. Note: Not all attributes shown in the diagram!



```
CNavigationSystem
-m_GPSSensor:CGPSSensor
-m_route:CRoute
-m_PoiDatabase:CPoiDatabase
-m_WpDatabase:CWpDatabase
+CNavigationSystem()
+run():void
-enterRoute():void
-printRoute():void
```

```
CWaypoint
-m_name:string
-m_latitude:double
-m_longitude:double
-m_type:wp_type
+CWaypoint(name:string,latitude:double,longitude:double,type:wp_type)
+getName():string
+getLatitude():double
+getLongitude():double
+print(format:int):void
```

```
CRoute
+CRoute()
+~CRoute()
+connectToPoiDatabase(pPoiDB:CPoiDatabase*):void
+connectToWpDatabase(pWpDB:CWpDatabase*):void
+addWaypoint(name:string):void
+addPoi(namePoi:string,afterWp:string):void
+print():void
```

```
CPOI
-m_type:t_poi
-m_description:string
+print(format:int):void
+CPOI(type:t_poi,name:string,description:string,latitude:double,longitude:double)
+~CPOI()
```

```
CPoiDatabase
+CPoiDatabase()
+addPoi(poi:CPOIconst &):void
+getPointerToPoi(name:string):CPOI*
```

```
CWpDatabase
+CWpDatabase()
+addWaypoint(wp:CWaypointconst &):void
+getPointerToWaypoint(name:string):CWaypoint*
+print():void
```

```
CGPSSensor
+CGPSSensor()
+getCurrentPosition():CWaypoint
```

You decide to use STL containers for storing the waypoints, points of interests and the route. Which container types are the best choices considering the following requirements:

- The waypoints and poi's are identified by their name
- When the navigation system calculates a route, first the waypoints will be added and afterwards the poi's will be inserted (at the appropriate places as shown in the sketch above)

Class	Attribute declaration (e.g. vector<int>)	Explanation of your choice
`CRoute`		
`CWpDatabase`		
`CPoiDatabase`		

One of the algorithms you have to implement is `addPoi()`. The poi with the name `namePoi` shall be inserted **<u>after the last</u>** waypoint having the name `afterWp`.

An extension of `addPoi()` is the += operator, which takes a string representing the name of a POI or waypoint, searches a corresponding POI and/or waypoint in the databases and adds them to the route. Hint: Use the already implemented methods as activities. If both a waypoint and a POI are found, the waypoint will be added before the POI.

Draw 2 activity diagrams describing the algorithms.

(Space for the addPoi and += operator activity diagram)

1.2 Polymorphic route implementation

1.2.1 Basic changes

Before implementing the `CWpDatabase`, we improve the design of the existing `CPoiDatabase`.

a) In the `CPoiDatabase`, we want to use storage space for POIs in such a way that it adapts to the numer of POIs stored automatically. The key idea is to use the POI's name as key and the POI as value. Replace the existing array with the appropriate STL container. Make sure that the interface does not change, i.e. that the method `getPointerToPOI` still returns 0 if a POI with the given name does not exist.

b) Test the class `CPoiDatabase` to make sure, that the complete system still works.

c) Now create a new class `CWpDatabase`. Although it is not recommended ;-), you may copy and paste some code from `CPoiDatabase`.

In the next step, we are going to modify the route to a polymorphic design.

Note: CWaypoint::m_type is not explicitly needed (and may be omitted). If you keep it in the design, it will identify if a waypoint is a waypoint or a POI. It can be used for debugging purposes.

d) In `CRoute`, remove the dynamic arrays containing the waypoints and pointers to POI's, they are not needed anymore. Remove all no longer required attributes.

e) Create an new dynamic array in `CRoute` with the appropriate STL storage container, containing CPOI or CWaypoint pointers (Polymorphism).

f) Modify the methods `addWaypoint()` and `addPoi()` to comply to the new interface and data structure. Note for `addPoi()`: The Poi with the name `namePoi` shall be inserted **after the last** waypoint having the name `afterWp`.

g) Provide a copy-constructor and assignment operator, implementing deep copies of a route. Note: Before implementing, check the functionality of the corresponding STL operators.

h) Provide tests as needed to verify all changes. Note: make sure, that all user errors (e.g. provision of wrong data) are handled!

Note: The print method will be modified in the next section. As in intermediate step, to avoid compilation problems, simply comment out all code.

1.2.2 Additional algorithms

i) Provide overloaded `<< operators` for `CWaypoint` and `CPoi`. Use the DEG, MIN, SEC format. Hint: Modify the scope of `CWaypoint` attributes and methods as needed.

j) Use the new << operators inside the print routine. This is kind of tricky, because the overloaded << operators are C-functions and the compiler does not provide a V-table entry for them.

- First let's try the good old print() method. Assuming, i is the iterator:
  `(*i)->print(2);`
  This should work without any problem.

- Now let's try the naïve approach with the overloaded operators:
  `cout << (**i) << endl;`
  What do you see? Explain (by adding a comment into the code)

- Obviously we have to tell the compiler, what kind of an object we have, either a Waypoint or a Poi. We can use the `dynamic_cast` operation for this:
  `dynamic_cast<CPOI*>(*i)`
  The cast will return the adress of the casted object (in this case of the `CPoi`-Object) in case the cast succeds, or `NULL` otherwise.

k) Provide an `operator += (string name)`, which takes a string representing the name of a POI or waypoint, searches a corresponding POI and/or waypoint in the databases and adds them to the end of the route. Hint: Use the already implemented methods. If both a waypoint and a POI are found, the waypoint will be added before the POI.

l) Provide the `operator +` for the `CRoute` class, which adds 2 routes by concatenating the content, if they are connected to the same waypoint and POI database. If not, an error message is shown and an empty route is returned.

1.3 Adding persistence

Up to now, you have filled the data bases using statements in your program. In a real system, of course, the data will be taken from persistent storage (e.g. database, file…).

Access to our persistent storage component shall be provided by the following interface:

```
class CPersistentStorage {

public:
       /**
        * Set the name of the media to be used for persistent storage.
        * The exact interpretation of the name depends on the implementation
        * of the component.
        *
        * @param name the media to be used
        */
       virtual void setMediaName(string name) = 0;

       /**
        * Write the data to the persistent storage.
        *
        * @param waypointDb the data base with way points
        * @param poiDb the database with points of interest
        * @return true if the data could be saved successfully
        */
       virtual bool writeData (const CWpDatabase& waypointDb,
                    const CPoiDatabase& poiDb) = 0;

       /**
        * The mode to be used when reading the data bases (see readData).
        */
       enum MergeMode { MERGE, REPLACE };

       /**
        * Fill the databases with the data from persistent storage. If
        * merge mode is MERGE, the content in the persistent storage
        * will be merged with any content already existing in the data
        * bases. If merge mode is REPLACE, already existing content
        * will be removed before inserting the content from the persistent
        * storage.
        *
        * @param waypointDb the the data base with way points
        * @param poiDb the database with points of interest
        * @param mode the merge mode
        * @return true if the data could be read successfully
        */
       virtual bool readData (CWpDatabase& waypointDb, CPoiDatabase& poiDb,
                    MergeMode mode) = 0;
};
```

The method setMediaName sets the name of the storage media to be used. The exact interpret-tation depends on the storage component. If the storage component uses a data base it could be the data base name. If it uses cloud storage, it could be a URL. In our first implementation, however, it is used as a file (base) name, see below. Note that persistence is provided by a component that exposes CPersistentStorage. As explained in the lecture about components and classes, this means that a persistence component provides a class that is derived from the interface CPersistentStorage and that you use an instance of this derived class to access the component's functionality. Of course, the component may need more classes than the class derived from the interface in order to implement the persistence functionality (though probably not in the case of our first implementation).

The first persistence component (you'll implement another one in the next exercise) uses two CSV files to store the data, one for each data base. CSV is an abbreviation for "Comma Separated Values". Using this format, the attributes of an object are written in a single line, by default separated using a comma. Despite the name "CSV", another character such as a semicolon is often used as separator. The important point is that the separator character used must not be part of the values of the attributes.

The file containing the way point data should be called "`<`*`media name`*`>-wp.txt`" and for example look like this:

```
Amsterdam;52.3731;4.8922
Darmstadt, 49.850,8.6527
```

The file with points of interest should be called "`<`*`media name`*`>-poi.txt`" and for example look like this:

```
RESTAURANT;Mensa HDA;The best Mensa in the world; 49.866934; 8.637911
SIGHTSEEING;Berlin; Berlin City Center; 52.51;13.4
```

a) Implement the persistence component with the behavior described above. In order to do this, it may be necessary to add methods to existing classes. Keep the changes to a minimum. Provide a hand written table with changes (table columns: "Class", "Change", "Reason").

b) Test the component by

- creating an instance of the persistence component in your main function,
- persisting your programmatically created data bases (and checking the content of the written files),
- replacing the code that creates the databases with an invocation of the readData method.

c) As the CSV-files may be edited by humans, it is possible that they contain lines that cannot be parsed due to format error made by the editor: there may, e.g. be too few fields (attributes) in a line. There may be text where a number is expected. A German editor may erroneously use a comma instead of the decimal point when specifying the latitude or longitude etc. Make sure that your implementation of readData detects as many errors as possible. An erroneous line should be reported on the console as:

    `Error:` *<problem>* `in line` *<number>*`:` *<line content>*

where *<problem>* is a hint such as "too few fields". Erroneous lines are skipped, reading continues with the next line.

Test your code by inserting erroneous lines in your CSV-files and starting your program again. Check that you get the expected error messages.