Exercise 3 – Navigation System (Advanced features)

In this exercise, you again 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)
- Make sure that all variables have the narrowest scope possible

Marking:

3.1	• Alternative persistence implementation	3
3.2	• Database template	3
3.3	• Systematic unit testing	3

Source, Internet

Please note that the allocation of time slots may change for the third 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.

3.1 Alternative persistence implementation

In the last exercise, you have implemented a persistence provider that used two CSV-files to store the data from the waypoint database and the poi database. In this exercise you're going to implement another persistence provider that uses a single file and the (nowadays) widely used JSON format for storing the data.

The JSON format basically consists of the list of comma separated attribute–value pairs of an object. The list is enclosed in curly brackets. A "value" is a string, number, boolean value, "null" or an object (i. e. another sequence of value–attribute pairs in curly brackets) or a list of values. You can find the precise definition of the format in RFC 7159 (https://tools.ietf.org/html/rfc7159).

A sample JSON representation for some waypoints and pois looks like this:

```
{
  "waypoints": [
    {
      "name": "Amsterdam",
      "latitude": 52.3731,
      "longitude": 4.8922
    },
    {
      "name": "Berlin",
      "latitude": 52.5166,
      "longitude": 13.4
    },
    {
      "name": "Darmstadt",
      "latitude": 49.85,
      "longitude": 8.6527
    }
  ],
  "pois": [
    {
      "name": "Berlin",
      "latitude": 52.51,
      "longitude": 13.4,
      "type": "Restaurant",
      "description": "Berlin City Center"
    },
    {
      "name": "Mensa HDA",
      "latitude": 10,
      "longitude": 20,
      "type": "Restaurant",
      "description": "The best Mensa in the world"
    },
    {
      "name": "Sitte",
      "latitude": 11,
      "longitude": 22,
      "type": "Restaurant",
      "description": "More expensive but also good"
    }
  ]
}
```

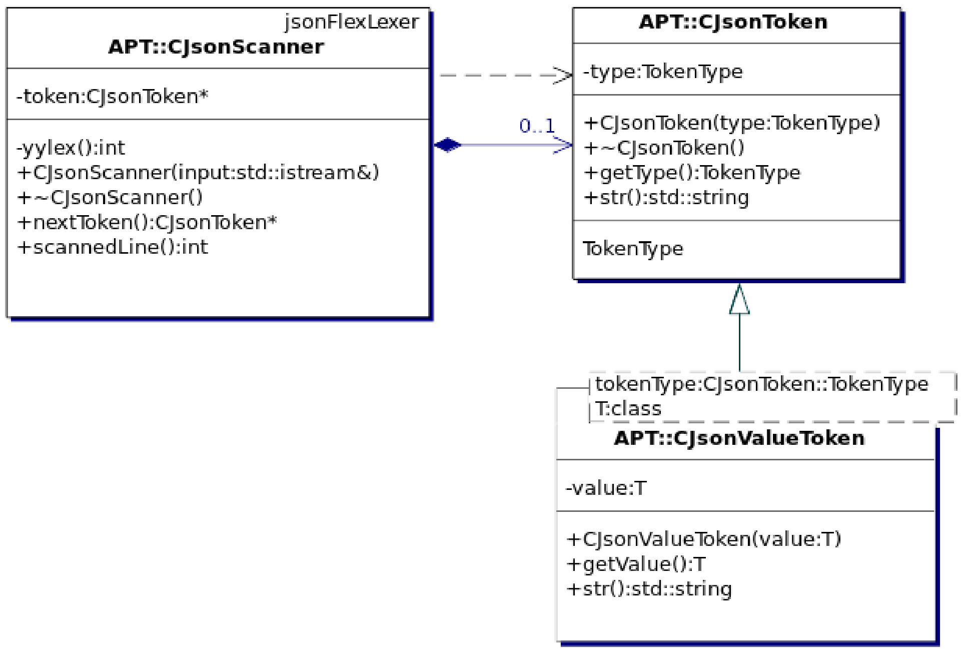
The new persistence provider (class name) is `CJsonPersistence`. The media name is the name of the (single) file that holds the JSON representation.

3.1.1 Writing the data

Implement `CJsonPersistence::writeData`. Make use of private functions and/or templates to ease your task and avoid duplicate code. The output should be pretty printed as in the example above.

3.1.2 Reading the data

In order to simplify reading the data, you get the implementation of a scanner (tokenizer) for JSON as class `CJsonScanner`. Download the required files from the moodle course and simply copy them into your `myCode` directory. The usage of the scanner is documented in detail in the header files. The following paragraph provides an overview of its function.



The scanner is created with an input stream as parameter. The input stream is expected to contain valid JSON, which is scanned (tokenized) by the scanner. Each time you invoke `nextToken()`, the scanner returns a pointer to an object of type `CJsonToken` (or a derived type). The object is managed internally by the scanner, so you need not (and must not) delete the token. The token type is `BEGIN_OBJECT`, `END_OBJECT`, `BEGIN_ARRAY`, `END_ARRAY`, `NAME_SEPARATOR`, `VALUE_SEPARATOR`, `STRING`, `NUMBER`, `BOOL` or `JSON_NULL`. For token types that have an associated value (`STRING, NUMBER`, `BOOL`) a specialized `CJsonValueToken` is returned. Carefully read the header file to understand how to use the tokens.

Your method `CJsonPersistence::readData` must implement a parser for the JSON representation. The parser continually requests tokens from the scanner and processes them until the input is exhausted.

You must implement the parser using a state machine.

First, draw a UML state diagram. Use the token types as events names. Some states that you should use (incomplete list): “Waiting for first token ” (or “Idle”), “Waiting for db name”, “Waiting for db begin”, “Waiting for attribute name”, “Waiting for name separator”, “Waiting for value” … Avoid duplications of states if they only differ due to a value. I. e. don't introduce a state “Waiting for longitude”. Rather use “Waiting for attribute name” and something like “store attribute name” as action in the transition that leads to the next state. Then, when you get the value token (event), you can use “assign value accoring to stored attribute name” as action.

Make sure that your parser works independent of the order of attributes. An input file is also valid if “latitude” comes before “name” or if the array of pois comes before the array of waypoints.

Now implement the parser using the switch(state) pattern as presented in the lecture.

When your parser parses your saved data properly, have a look at possible input errors. Imagine somebody providing the data as a file written in an editor. What can go wrong? Check for as many errors as possible (conider the state diagram to find possible problems). Report the line and the situation that causes the error on the console. Check your code by creating and reading files with errors.

The implementation of the scanner is currently very lax regarding invalid characters in the input. If you put an "@" somewhere in your input file (outside a string and surrounded by white space) the text will be parsed nevertheless. The invalid character is recognized in `CJsonScanner.cpp` line 33. The problem is how to report the invalid character. You could define another token type, but as an invalid character isn't a valid token, modeling it as a token doesn't really fit. To solve that problem, define an exception that is thrown by `nextToken()` if an invalid character is encountered. The exception should carry the required information for a proper error message (line number, invalid character found). Catch the exception when you invoke the method and produce an error message on the console as you do for the other errors.

3.2 Database templates

The waypoint and poi databases are rather similar. Entries can be added, searched for, they can be cleared etc. The main difference is the type of the entries. The waypoint database contains `CWaypoint` objects that can be looked up by name (the key value) and the poi database contains `CPOI` objects (that can also be found by name).

In order to simplify your code and avoid duplication, provide a template class for a database that allows you to choose the type of the key value (string and int should at least be properly supported) and the type of the contained values (arbitrary types allowed, as long as they are supported by `std::map`).

Think carefully about appropriate names for the methods of your template class. The names should be “neutral”, i. e. good matches for all usages of the template class.

Now use the template classes in your implementation. In order to avoid time consuming renames, derive the classes that you have used so far (`CWpDatabase` and `CPoiDatabase`) from your template class specifying the appropriate type for the database's content. In the derived class, add the “old” methods with implementations that simply invoke the appropriate method from the template class.

3.3 Systematic unit testing

Add a method “`const vector<const CWaypoint*> getRoute()`” to class `CRoute`. The method returns a list of pointers to the waypoints and POIs that have been added to the route in the same order as they are stored in the route. (Note that due to the usage of the `const`-qualifiers, it may be necessary to change some “ordinary” methods to `const`-methods in your implementation.)

Implement exhaustive unit test cases for class `CRoute` using the cppunit library introduced in the lecture. An approach how to separate the “production” `main` from the “testing `main`” has been shown in the lecture. To make the task easier, you may, however, alternatively copy all source files to a new project and write a “main” for executing the unit tests.

To structure the unit tests, create one test suite for each API function (method). Within that suite, provide tests for the different possible behaviors of that method. (The most complex method to test is certainly “`addPoi`”.)

Try to consider all possible situations when writing your test cases. For example what happens if you call `addPoi` without first connecting to the poi database? You may find that the specification is incomplete, i.e. the behavior for some of these problematic cases isn't specified. Note those cases down in a list and describe the behavior that you have implemented. Implement a behavior for the problematic cases that you'd expect from a good design. Consider all methods shown in the lecture about error handling (special return values, throwing exception).

As a criterion for the completeness of your tests, measure the code coverage and show your results during the review. Try to achieve 100% coverage.