Lab 10 – Reflection

Es gibt viele Eigenschaften, die eine gute Route auszeichnen können. Daher existieren viele verschiedene Routen-Algorithmen, die verschiedene Aspekte der Berechnung optimieren. Das Routenplanner-Programm soll deshalb so angepasst werden, dass mittels Konfiguration bestimmt werden kann, welcher implementierter Algorithmus zur Ausführung kommt. Dazu wird ein Interface eingeführt, welches die jeweiligen Algorithmen implementieren müssen; die konkreten Implementierungen sollen im Routenplanner Code nicht mehr bekannt sein. D.h. die eigentlichen Algorithmen sollen in externen Bibliotheken vorhanden sein und dynamisch geladen werden können.

Aufgabe 1 – Dynamisches Laden des Algorithmus

a) Führen Sie zunächst das folgende Interface (als `ILinks.cs`) ein und lassen Sie Ihre `Links`-Klasse dieses Interface implementieren.

```
public interface ILinks
{
    /// <summary>
    /// Reads a list of links from the given file.
    /// Reads only links between existing cities.
    /// </summary>
    /// <param name="filename">name of input file</param>
    /// <returns>number of links read</returns>
    int ReadLinks(string filename);

    /// <summary>
    /// Determines the shortest path between the given cities.
    /// </summary>
    /// <param name="fromCity">name of start city</param>
    /// <param name="toCity">name of destination city</param>
    /// <param name="mode">transportation mode</param>
    /// <returns></returns>
    List<Link> FindShortestRouteBetween(string fromCity, string toCity,
                                        TransportMode mode);
}
```

Legen Sie die neue Datei im RouteplannerLib Verzeichnis ab.

b) Achten Sie darauf, dass Sie nur noch das Interface im Code referenzieren, die konkreten Klassen sollen nicht mehr vorkommen.

Wichtig: Das gilt nur für den Applikations-Code; verändern Sie **NICHT** den Testcode.

Note: Es sollten im Code eh keine Referenzen vorhanden sein.

c) Führen Sie nun im Projekt `RoutePlannerLib` eine Factory-Klasse ein, die eine entsprechende `ILinks`-Implementierung über einen gegebenen Namen instanziert und zurückgibt. Die Factory-Klasse soll eine Create-Methode anbieten, die den Klassen-Namen als Parameter entgegen nimmt:

```
public class LinksFactory
{
    static public ILinks Create(Cities cities, string algorithmClassName)
    {
        //TODO: Create Links-class with the supplied name
    }
}
```

Wird der Name einer ungültigen Klasse spezifiziert, soll eine `NotSupportedException` geworfen werden. Suchen Sie den angegebenen Typen nicht nur in einem einzelnen Assembly, sondern in allen aktuell geladenen Assemblies, mit Hilfe der Methode `AppDomain.CurrentDomain.GetAssemblies()`.

Hinweis: Die Trennung ist damit noch nicht ganz komplett. Zur Vervollständigung müsste noch die konkrete Links-Implementierung in eine eigene Assembly ausgelagert werden.

d) Testen Sie Ihre Implementierung mit den Unit-Tests im Files «`LinksFactoryTest_Lab10.cs`» im «`caseStudyFiles`» Ordner.

Aufgabe 2 – Serialisierung/Deserialierung mittels Reflection

Eine etwas anspruchsvollere Reflection Aufgabe

a) Sie sollen nun die eingelesenen Städte mit einem eigenen Serializer/Deserializer persistieren. Schreiben Sie einen generischen Serializer, der in der Lage ist beliebige Klassen und deren öffentlichen Properties vom Typ `string`, `double` und `int` mittels Reflection zu serialisieren und deserialisieren. Das Format der Serialisierung finden Sie im Anhang.

   Legen Sie dazu die neuen Klassen `SimpleObjectReader` und `SimpleObjectWriter` im Namespace `Fhnw.Ecnf.RoutePlanner.RoutePlannerLib.Util` an. Das Interface der Klassen können Sie aus den Unittests ableiten.

b) Erweitern Sie die Serialisierung so, dass eigene Datentypen unterstützt werden (z.B. die Property `Location` vom Typ `WayPoint`). Auch hier sollen Sie alle geladenen Assemblies berücksichtigen.

   Testen Sie Ihre Lösung mit zusätzlichen Test File `Lab10DeSerializerTest.cs`.

Hinweise:

- *Für die Deserialisierung müssen Sie die Klassen City und WayPoint um einen Default-Konstruktor ergänzen, sowie für alle Properties auch die Setter-Methode anbieten.*
- *Es geht bei dieser Aufgabe **nicht** um das `[Serializable]`-Attribut. Sie sollen nicht bestehende .NET Serialisierungs-Technologien einsetzen, sondern die nötige Infrastruktur mittels Reflection selbst entwickeln.*
- *Sie sollen die `Next`-Methoden rekursiv implementieren, damit «nested Objects» korrekt behandelt werden.*
- *Verwenden Sie `Assembly.GetType(string)` damit auch Typen gefunden werden, welche `internal` markiert sind.*

Format: für ein serialisiertes Objekt (BNF[1]-Format)

<object> ::=
"Instance of" <FullQualifiedName>"\r\n"{<Property>}"\r\nEnd of instance\r\n"

<Property> ::= [<PropertyName>=<PropertyValue>|
<PropertyName>" is a nested object..."
<Object>]

<PropertyValue> ::= [‘"’<text>‘"’|<number>]

Hinweis: Die Properties sollen in alphabetischer Reihenfolge serialisiert werden.

Beispiel-Serialisierung eines City-Objekts

```
Instance of Fhnw.Ecnf.RoutePlanner.RoutePlannerLib.City
Country="Switzerland"
Location is a nested object...
Instance of Fhnw.Ecnf.RoutePlanner.RoutePlannerLib.WayPoint
Latitude=1.1
Longitude=2.2
Name="Aarau"
End of instance
Name="Aarau"
Population=10
End of instance
```

[1] https://de.wikipedia.org/wiki/Backus-Naur-Form