Praktikumsaufgabe 3

Im dritten Praktikumsversuch lernen Sie den Umgang mit *Templates* sowie die *Serialisierung von Objekten* (eine Abbildung von strukturierten Daten in eine sequentielle Form) mithilfe des JSON-Formats und den von Qt hierfür bereitgestellten Klassen kennen. Insbesondere erfordert dies auch die Einarbeitung in eine neue *API (Application Programming Interface)*.

Hauptsächlich wird die Serialisierung zur *persistenten Speicherung* (Speicherung von Daten über das Programmende hinaus) eingesetzt. In dem Zusammenhang wird oft auch der Begriff *Marshalling* verwendet, welches im Gegensatz zur Serialisierung auch die architektur-, sprach- und plattform-unabhängige Darstellung von Objekten beliebigen Typs garantiert.
In TI3:Programmierung wurde für die persistente Speicherung neben einer binären Serialisierung mit dem Modul `pickle` auch eine textuelle Serialisierung mit der Hilfe von XML verwendet.
In neuerer Zeit wird für die plattform-unabhängige Speicherung und die Übertragung von Daten (besonders in Web-Anwendungen oder mobilen Apps) XML durch *JSON (JavaScript Object Notation)*, *YAML (YAML Ain't Markup Language)* oder *Google's Protocol Buffers* abgelöst.
Die Syntax von JSON ist sehr viel einfacher gestaltet und ist daher leichter zu lesen und zu schreiben. In der Regel produziert JSON auch einen geringeren Overhead im Vergleich zu XML mit seinen länglichen ausgeschriebenen Tag-Namen.
Weiterführende Informationen zum Vergleich der unterschiedlichen Serialisierungsformate (in Qt) finden Sie z. B. unter:

`https://blog.qt.io/blog/2018/05/31/serialization-in-and-with-qt/`
`https://en.wikipedia.org/wiki/Comparison_of_data-serialization_formats`

I) Projekterstellung:

1. **Hauptprogramm main:**

   Da in diesem Praktikum Klassen von Qt verwendet werden, ist es am einfachsten, wenn Sie sich für dieses Praktikum eine **Qt Konsolenanwendung** als Projekt erstellen (denn dann müssen Sie die automatisch generierte Qt Projekt-Datei nicht mehr hinsichtlich Include-Dateien und Bibliotheken anpassen).

   Hierfür wählen Sie im Menue

   ```
   Datei->Neu->Anwendung->Qt Konsolenanwendung
   ```

   aus.

   Ersetzen Sie das dadurch erstellte Hauptprogramm mit der `main`-Funktion aus der Praktikumsaufgabe 2.

   (Auf die Verwendung von `QCoreApplication` kann in dieser Aufgabe verzichtet werden.) Die Include-Anweisungen müssen dann später ebenfalls angepasst werden.

2. Kopieren Sie nun die anderen Header- und Quellcode-Dateien des zweiten Praktikums in das neue Projektverzeichnis. **Fügen Sie diese Dateien aber noch nicht zum neuen Projekt hinzu (s. u.).**

II) Verallgemeinerung der Klasse DeList:

1. Ziel ist es zunächst, dass die Klasse `DeList` nicht nur zur Ablage von `IKomponente*` dient, sondern Objekte beliebiger Klassen und Typen darin abgelegt werden können. Dazu muss die Klasse in ein **Klassen-Template** umgewandelt werden. Damit das möglich wird, müssen auch die Klasse `Iterator` und die Struktur `IKomponentenElement` angepasst werden:

   a) Die Struktur `IKomponentenElement` soll in `TElement` umbenannt werden, deshalb benennen Sie die Datei in `telement.h` um.

   b) Sie können nun nach der obigen Umbenennung der Header-Datei alle Dateien, die Sie neu in das Projektverzeichnis kopiert haben, ihrem Projekt hinzufügen.

   c) Editieren Sie nun die Header-Datei `telement.h`, benennen die Struktur in `TElement` um und passen Sie auch den Include-Guard (d. h. die kapselnden Präprozessordirektiven `#ifndef` und `#endif`) entsprechend an.

   d) Wandeln Sie nun diese Struktur in ein Template mit Typ-Parameter `T` um. In der Struktur ist also der Typ `IKomponente*` durch den Typ-Parameter `T` zu ersetzen.

   e) Wandeln Sie nun die Klasse `Iterator` in ein Template mit Typ-Parameter `T` um.
   Hinsichtlich der Methoden, die in der Datei `iterator.cpp` definiert (implementiert) sind gibt es 2 Möglichkeiten:

      i) Sie können alle Definitionen der Methoden in die zugehörige Header-Datei `iterator.h` nach der Deklaration der Klasse `Iterator` einfügen und entfernen dann die Datei `iterator.cpp` aus Ihrem Projekt.

      ii) Sie belassen die Definition dieser Methoden in der Datei `iterator.cpp` **ohne** sie in die zugehörige Header-Datei `iterator.h` zu kopieren. Benennen Sie die Datei `iterator.cpp` in `iterator_impl.h` um. Ergänzen Sie die Datei um einen geeigneten Include-Guard.
      Überall dort im Programm, wo sie eine konkrete Instanz dieses Templates benötigen, müssen Sie nur die Header-Datei `iterator_impl.h` inkludieren.

   f) Wandeln Sie zuletzt noch die Klasse `DeList` in ein Template mit Typ-Parameter `T` um.
   Hinsichtlich der Methoden, die in der Datei `deList.cpp` definiert (implementiert) sind gibt es ebenfalls, die schon oben beschriebenen beiden Möglichkeiten:

      i) Sie können alle Definitionen der Methoden in die zugehörige Header-Datei `deList.h` nach der Deklaration der Klasse `DeList` einfügen und entfernen dann die Datei `deList.cpp` aus Ihrem Projekt.

      ii) Sie belassen die Definition dieser Methoden in der Datei `deList.cpp` **ohne** sie in die zugehörige Header-Datei `deList.h` zu kopieren. Benennen Sie die Datei `deList.cpp` in `deList_impl.h` um. Ergänzen Sie die Datei um einen geeigneten Include-Guard.
      Überall dort im Programm, wo sie eine konkrete Instanz dieses Templates benötigen, müssen Sie nur die Header-Datei `deList_impl.h` inkludieren.

2. Passen Sie nun auch die Klasse `Werkstueck` so an, dass sie nur noch die Template-Instanziierungen `DeList<IKomponente*>` und `Iterator<IKomponente*>` verwendet.

   Danach sollte Ihr Hauptprogramm `main()` ausführbar sein, so dass Sie die obigen Änderungen überprüfen können. Die Ausführung des Hauptprogramms soll die gleiche Ausgabe wie im Praktikum 2 liefern.

III) Serialisierung zur persistenten Sicherung:

1. Die Serialisierung soll mithilfe von JSON und den von Qt hierfür bereitgestellten Klassen `QJsonObject`, `QJsonArray`, `QJsonDocument` erfolgen.

   a) Informieren Sie sich zunächst über *JSON (JavaScript Object Notation)*, z.B. im Web unter `http://json.org/`.

   b) Studieren Sie **gründlich** die von Qt bereitgestellte API für die Verwendung von JSON, z.B. im Web unter:

   ```
   https://doc.qt.io/qt-5/json.html
   https://doc.qt.io/qt-5/qtcore-serialization-savegame-example.html
   https://doc.qt.io/qt-5/qjsonobject.html
   https://doc.qt.io/qt-5/qjsonarray.html
   https://doc.qt.io/qt-5/qjsondocument.html
   ```

   c) Die Serialisierung des `Werkstueck`-Objektes `w0` in Ihrem Hauptprogramm soll die folgende JSON-Datei erzeugen:


```
{
    "components": [
        {
            "diameter": 8,
            "hasParent": true,
            "type": "Bohrung",
            "x": 1,
            "y": 2
        },
        {
            "angle": 0.7853981633974483,
            "diameter": 5,
            "hasParent": true,
            "length": 2.8284271247461903,
            "type": "Fraesung",
            "x": 2,
            "y": 3
        },
        {
            "diameter": 4,
            "hasParent": true,
            "type": "Bohrung",
            "x": 2,
            "y": 6
        },
        {
            "angle": 0,
            "diameter": 6,
            "hasParent": true,
            "length": 3,
            "type": "Fraesung",
            "x": 6,
            "y": 7
        },
        {
            "components": [
                {
                    "diameter": 6,
                    "hasParent": true,
                    "type": "Bohrung",
                    "x": 1,
                    "y": 2
                },
                {
                    "angle": 2.6779,
                    "diameter": 3,
                    "hasParent": true,
                    "length": 2.23606797749979,
                    "type": "Fraesung",
                    "x": 3,
                    "y": 3
                },
                {
                    "diameter": 8,
                    "hasParent": true,
                    "type": "Bohrung",
                    "x": 5,
                    "y": 1
                },
                {
                    "components": [
                        {
                            "angle": -0.7853981633974483,
                            "diameter": 4,
                            "hasParent": true,
                            "length": 1.4142135623730951,
                            "type": "Fraesung",
                            "x": 1,
                            "y": 2
                        },
                        {
                            "diameter": 8,
                            "hasParent": true,
```

```
                    "type": "Bohrung",
                    "x": 5,
                    "y": 2.5
                },
                {
                    "diameter": 6,
                    "hasParent": true,
                    "type": "Bohrung",
                    "x": 6,
                    "y": 0.5
                }
            ],
            "hasParent": true,
            "height": 3,
            "pathIsOptimized": true,
            "type": "Werkstueck",
            "width": 10,
            "x": 6,
            "y": 3
        }
    ],
    "hasParent": true,
    "height": 7,
    "pathIsOptimized": true,
    "type": "Werkstueck",
    "width": 18,
    "x": 6,
    "y": 10
},
{
    "diameter": 7,
    "hasParent": true,
    "type": "Bohrung",
    "x": 10,
    "y": 8
},
{
    "diameter": 5,
    "hasParent": true,
    "type": "Bohrung",
    "x": 7,
    "y": 2
},
{
    "angle": 3.9269908169872414,
    "diameter": 4,
    "hasParent": true,
    "length": 5.656854249492381,
    "type": "Fraesung",
    "x": 17,
    "y": 6
}
],
"hasParent": false,
"height": 18,
"pathIsOptimized": true,
"type": "Werkstueck",
"width": 25,
"x": 0,
"y": 0
}
```


d) Um die Implementierung einfach zu halten, wird im Folgenden jede Klasse selber dafür verantwortlich sein, ihre Daten in einem `QJsonObject` zu speichern und dieses an die Außenwelt zurück zu geben

Dies wird dadurch umgesetzt, dass in der Schnittstelle `IKomponente` die **rein virtuelle Methode**

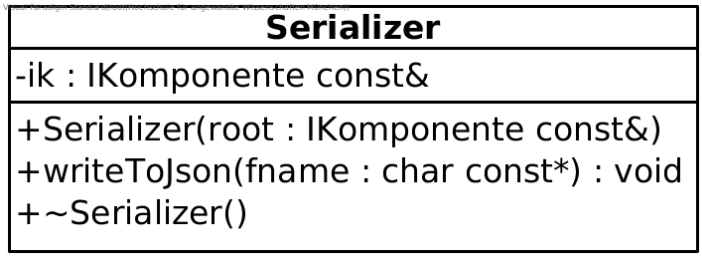
```
virtual QJsonObject toJson() const = 0;
```

neu hinzugefügt wird.

Damit muss in allen Unterklassen diese Methode geeignet implementiert werden.

Welche Vor- und Nachteile ergeben sich durch diese Art der Umsetzung. (Was müsste man z.B. umsetzen, wenn auch eine Serialisierung nach **XML** oder **YAML** gewünscht ist? Welche Alternative gäbe es noch?)

e) Fügen Sie nun dem Projekt eine Klasse `Serializer` mit dem im folgenden UML-Klassendiagramm angegebenen Elementen hinzu.



Die Methode `writeToJson(const char* fname) const` schreibt mit der Hilfe eines `QJsonDocument`-Objektes das Objekt, auf das die Referenz `ik` verweist, in eine Datei mit dem File-Namen `fname`.

Sollte die Datei nicht geöffnet werden können, ist eine Standard-Exception vom Typ `fstream::failure` mit der Fehler-Meldung
`"Datei konnte nicht geoeffnet werden."` zu werfen.

Tritt während des Schreibens ein Fehler auf, so ist ebenfalls eine `fstream::failure`-Exception mit der Fehler-Meldung
`"Datei konnte nicht beschrieben werden."` zu werfen.

f) Ergänzen Sie nun im Hauptprogramm alle notwendigen Anweisungen innerhalb eines passenden `try-catch`-Blocks, um das `Werkstueck`-Objekt `w0` mit der Hilfe eines Objektes vom Typ `Serializer` in die Datei `w0.json` zu serialisieren.

Vergleichen Sie den Inhalt dieser Datei mit dem oben angegebenen Listing der JSON-Datei.

2. (optional) **Reverse-Engineering mit dem CASE-Tool Visual Paradigm:** Erstellen Sie mit der Hilfe des CASE-Tools Visual Paradigm ein UML-Klassendiagramm für das gesamte Praktikumsprojekt und speichern Sie es als PDF-Datei ab.