

Hinweis zur Aufgabe 1 c) des Übungsblattes *Sortieralgorithmen*

Das Rüstzeug

Das Rüstzeug: Code für ein Qt-Projekt

Uns werden Code-Bruchstücke zur Verfügung gestellt, von denen über die Aufgabe a) bekannt ist, dass dieser Code Fehler enthält.

Unser wichtiges Rüstzeug heißt: * Code lesen; sich seinen Informationsgehalt erschließen; die Fehler entdecken.

Typischerweise ist es nicht möglich beim einmaligen Lesen alle Information zu erfassen. Wie ein mehrstufiger Leseprozess aussieht, ist stark vom individuellen Vorwissen abhängig.

Trotzdem ist es sinnvoll, sich allgemeine Leitregeln zum Code-Lesen bewusst zu machen.

- Die Regel zum erste Code-Lesen:
Sich einen Überblick verschaffen.
- Die Regel zum zweiten Code-Lesen:
Klären wie vorhandener Code für die eigene Aufgabe wiederverwendet werden kann.

Das erste Code-Lesen

Das erste Lesen verschafft sich einen Überblick.

Beim ersten Lesen hat man die folgenden Fragen im Hinterkopf:

- Was macht ein Code-Abschnitt?
- Wofür wird der Code-Abschnitt gebraucht?

Das Lesen von Funktionen

Code-Abschnitte, die Funktionen darstellen, lassen sich in einen Kopf-Bereich und einen Definitionsbereich oder Körperbereich (*engl.* body) unterteilen.

Die zuverlässigsten Antworten auf die Frage *Was macht ein Code-Abschnitt?* bekommt man aus dem Kopf eines Code-Abschnittes.

- aus den Namen
- aus einem Kommentar zum Kopf
- aus den übergebenen Parameter

Wenn der Kopfbereich eines Elementes nicht genug information liefert, kann es hilfreich sein den Body zu studieren.

Der (fehlerhafte) Code

Die (fehlerhaften) Code-Bruchstücke für ein Qt-Applikation Sortieralgorithmen

```
typedef int datentyp; // Datentyp des Arrays

datentyp* origArray;

void arrayAnzeigen(datentyp *array, int anz, QListWidget* lwAnzeige)
{
    lwAnzeige->clear();
    lwAnzeige->setVisible(false); //sonst kostet die Anzeige zu viel Rechenzeit!
    for (int i=0; i<=anz; i++) {
        lwAnzeige->addItem(QString::number(array[i]));
    }
    lwAnzeige->setVisible(true);
}

void bubbleSort(datentyp *feld, int anz) //aufsteigendes Sortieren
{
    datentyp *tmp;
    for (int x=0; x<anz-1; x++) {
        for (int i=0; i<=anz-1; i++) {
            if (feld[i]<feld[i+1]) {
                tmp = feld[i];
                feld[i+1] = feld[i];
                feld[i+1] = tmp;
            } //if
        } //for i
    } // for x
}

void FrmMain::on_btnBubbleSort_clicked()
{
    this->setCursor(Qt::WaitCursor);
    QTime time;
    datentyp* tmpArray = new datentyp[anzElemente];
    tmpArray = origArray;
    time.start();
    bubbleSort(tmpArray, anzElemente);
    ui->lblZeitBubbleSort->setText(QString::number(time.elapsed())+ " ms");
    arrayAnzeigen(tmpArray, anzElemente, ui->lwKopie);
    delete tmpArray;
    this->setCursor(Qt::ArrowCursor);
}
```

Allgemeines Vorwissen

C++ erlaubt die Benutzung von globalen Funktionen.

NOTE

C++ ist keine rein objektorientierte Sprache. Hingegen ist Java eine rein objektorientierte Sprache, in der weder globale Methoden noch globale Variablen existieren.

Java ist eine Sprache, in der weder globale Methoden noch globale Variablen existieren.

Siehe dazu auch:

- Java lernen in abgeschlossenen Lerneinheiten. *Springer Verlag*

NOTE

Das Kapitel 11 *Wie erstelle ich objektorientierte Programme* enthält im Abschnitt *Statische Elemente* auf der Seite 148. Statische Elemente einer Klasse.

Die statischen Elemente einer Klasse, also Klassenvariablen und Klassenmethoden, übernehmen in Java die Aufgaben von globalen Elementen.

Die Erkenntnisse beim ersten Code lesen

Unser Beispiel Quellcode, den wir oben zeigen, enthält eine globale Variablendeklaration, zwei globale Funktionen und eine Funktion, die zum Namensraum `FrmMain` gehört.

Die globale Deklaration der Variablen.

```
typedef int datentyp; // Datentyp des Arrays  
  
datentyp* origArray;
```

Die globale Funktion `arrayAnzeigen`

```
void arrayAnzeigen(datentyp *array, int anz, QListWidget* lwAnzeige)
```

und

Die globale Funktion `bubbleSort`

```
void bubbleSort(datentyp *feld, int anz)
```

Außer diesen globalen Elementen enthält der Code die Funktion `FrmMain::on_btnBubbleSort_clicked()`.

Für jedes dieser Bruchstücke müssen wir uns die Fragen zur ersten Lese-Runde beantworten:

- Was macht der Code-Abschnitt?
- Wofür wird der Code-Abschnitt gebraucht?

FrmMain::on_btnBubbleSort_clicked(): Funktion eines Namensraum oder Instanzmethode einer Klasse?

Es gibt zwei Interpretationsmöglichkeiten für diese Funktion.

1. Sie gehört zum Namensraum (*engl.* namespace) `FrmMain`

In diesen Fall wäre diese eine Funktion, die gekapselt wird durch den Namensraum `FrmMain`.

2. Sie gehört zur Klasse `FrmMain` und ist eine Instanzmethode der Klasse

Der verwendete Stilguide erfordert:

```
class FrmMain : public QMainWindow
```

Mit anderen Worten: Die Klasse `FrmMain` leitet sich von der Klasse `QMainWindow` ab. Die Klasse `FrmMain` repräsentiert als das Hauptfenster der Anwendung.

Somit gilt:

Die Methode `on_btnBubbleSort_clicked()` gehört zur Klasse `FrmMain`.

```
void FrmMain::on_btnBubbleSort_clicked()
```

Methoden sind Funktion, die zu einer Klasse gehören.

Eine Funktion, die zu einer Klasse gehört, wird als *Methode* bezeichnet.

Es wird unterschieden zwischen *Klassenmethoden* und *Instanzmethoden*.

NOTE

- Bestimme, ob es sich bei `FrmMain::on_btnBubbleSort_clicked()` um eine Instanz- oder eine Klassenmethode handelt.
- Erläutere den Unterschied zwischen einer Instanz- oder einer Klassenmethode.

Das Modellieren von globalen Elementen in UML

UML unterstützt die Modellierung von globalen Elementen nur indirekt.

Reine OO-Sprachen kennen keine globalen Elemente. UML wurde für reine OO-Sprachen entwickelt. Daher bietet UML keine direkte Modellierung für globale Elemente an.

Indirekt lassen sich globale Elemente modellieren. Zwei Modellierungsarten für globale Elemente sind:

- Die Modellierung von globalen Elementtypen in UML als Objekttypen
- Die Modellierung globaler Elemente in UML als Klassenelemente

Die Modellierung von globalen Elementtypen in UML als Objekttypen

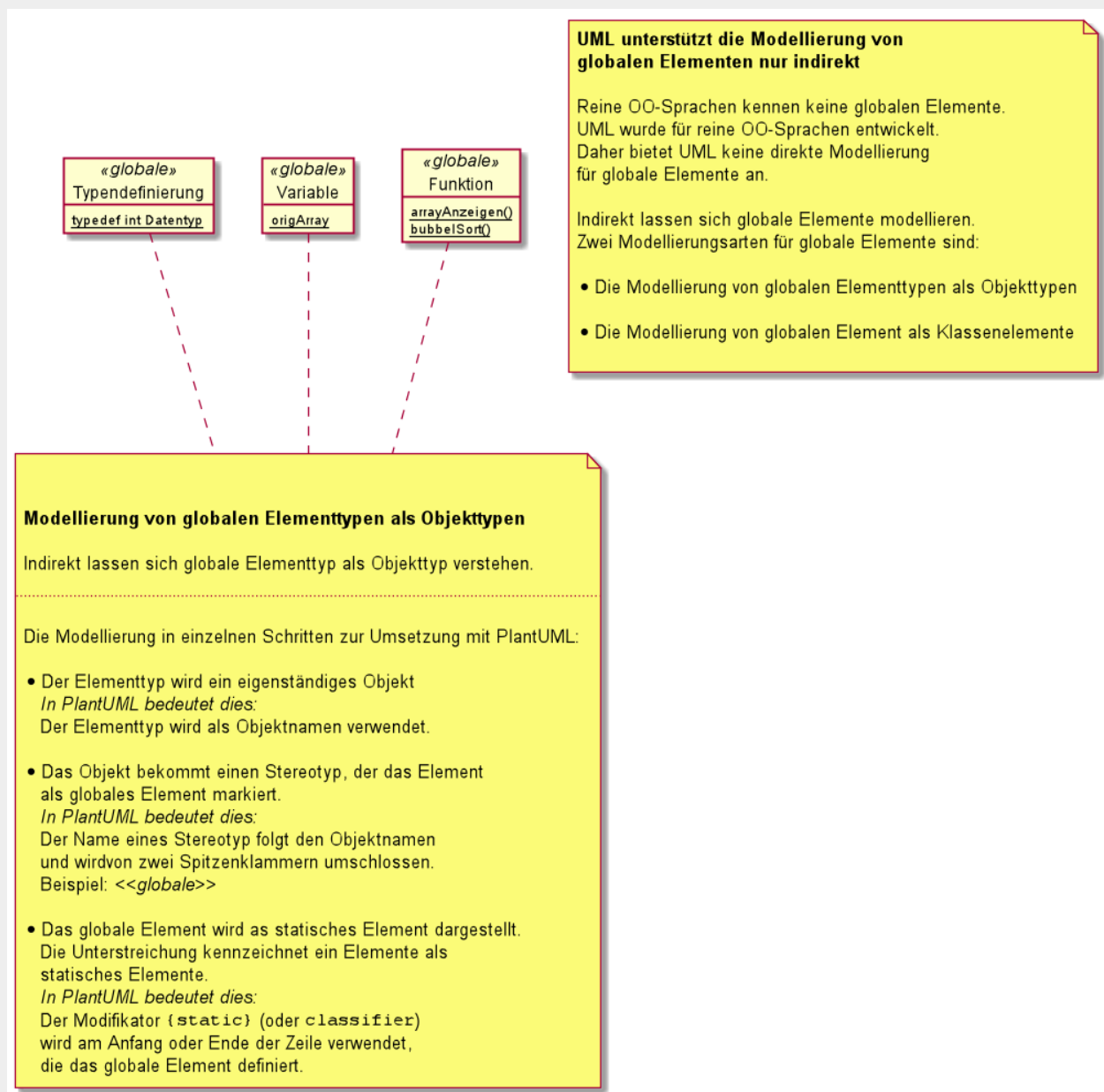
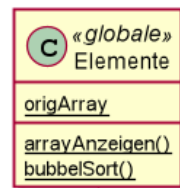


Figure 1. Die Modellierung von globalen Elementtypen in UML als Objekttypen

Die Modellierung globaler Elemente in UML als Klassenelemente



UML unterstützt die Modellierung von globalen Elementen nur indirekt

Reine OO-Sprachen kennen keine globalen Elemente. UML wurde für reine OO-Sprachen entwickelt. Daher bietet UML keine direkte Modellierung für globale Elemente an.

Indirekt lassen sich globale Elemente modellieren. Zwei Modellierungen für globale Elemente sind:

- Die Modellierung von globalen Elementtypen als Objekttypen
- Die Modellierung von globalen Element als Klassenelemente

Die Modellierung von globalen Element als Klassenelemente

Indirekt lassen sich globale Elemente zum Beispiel als *Klassenelemente* einer Klasse modellieren.

Hier wird der Klassenname `Elemente` verwendet mit dem Stereotype wie `<<globale>>`.

Mit der Unterstreichung der Elemente wird angezeigt, dass es keine Instanzelemente sind, sondern Klassenelemente.

In PlantUML werden Klassenelemente erzeugt, in dem der Modifikator (`static`) (oder `classifier`) am Anfang oder Ende der Zeile verwendet, die das Element definiert.

Figure 2. Die Modellierung der globalen Elementen in UML als Klassenelemente

Der Namespace

Globale Elemente sind im implizit vorhandenen globalen Namespace deklariert. C++ erlaubt neben den globalen Namespace weitere Namensräume zu definieren Namensräume verhindern Namenskollisionen, die in größeren Projekt leicht entstehen. C++ selbst benutzt den Namensraum `std::` für die Elemente, die von der C++-Standard Library benutzt werden.

Deklaration im globalen Namespace sind möglichst zu vermeiden.

Wenn ein Bezeichner nicht in einem expliziten Namespace deklariert ist, ist er Teil des impliziten globalen Namespaces. Vermeiden Sie im Allgemeinen, wenn möglich, Deklarationen im globalen Gültigkeitsbereich zu verwenden, mit Ausnahme der Einstiegspunkt-Hauptfunktion, die sich im globalen Namespace befinden muss. Um einen globalen Bezeichner explizit zu qualifizieren, verwenden Sie den Bereichsauflösungsoperator ohne Namen, wie in `::SomeFunction(x);`. Dadurch wird der Bezeichner von allen anderen Elementen gleichen Namens in allen anderen Namespaces unterschieden, und es vereinfacht außerdem das Verständnis Ihres Codes für andere.

— Erläuterung zum Namespace von C++ auf [doc.microsoft.com](https://docs.microsoft.com)

Reine OO-Sprachen kennen keine stand-alone Elemente

Weil C++ die Deklaration von Namespaces ermöglicht, gibt es in C++ nicht nur globale Elemente, sondern auch sogenannte allein-stehenden Elemente (*engl. standalone elements*) in Namensräumen.

Vor diesen Hintergrund müssen wir unsere Aussage "Reine OO-Sprachen kennen keine globalen Elemente" präziser fassen: *Reine OO-Sprachen kennen keine stand-alone Elemente*. Denn in einer reinen OO-Sprache werden alle Elemente durch Klassen gekapselt.

In C++ sind stand-alone Elemente möglich, die nicht durch eine Klasse gekapselt werden: Variablen, Konstanten oder auch Funktionen. Wohl aber können in C++ solche stand-alone Elemente durch Namensräume gekapselt werden.

Wir passen unser Diagramm dieser Erkenntnis an.

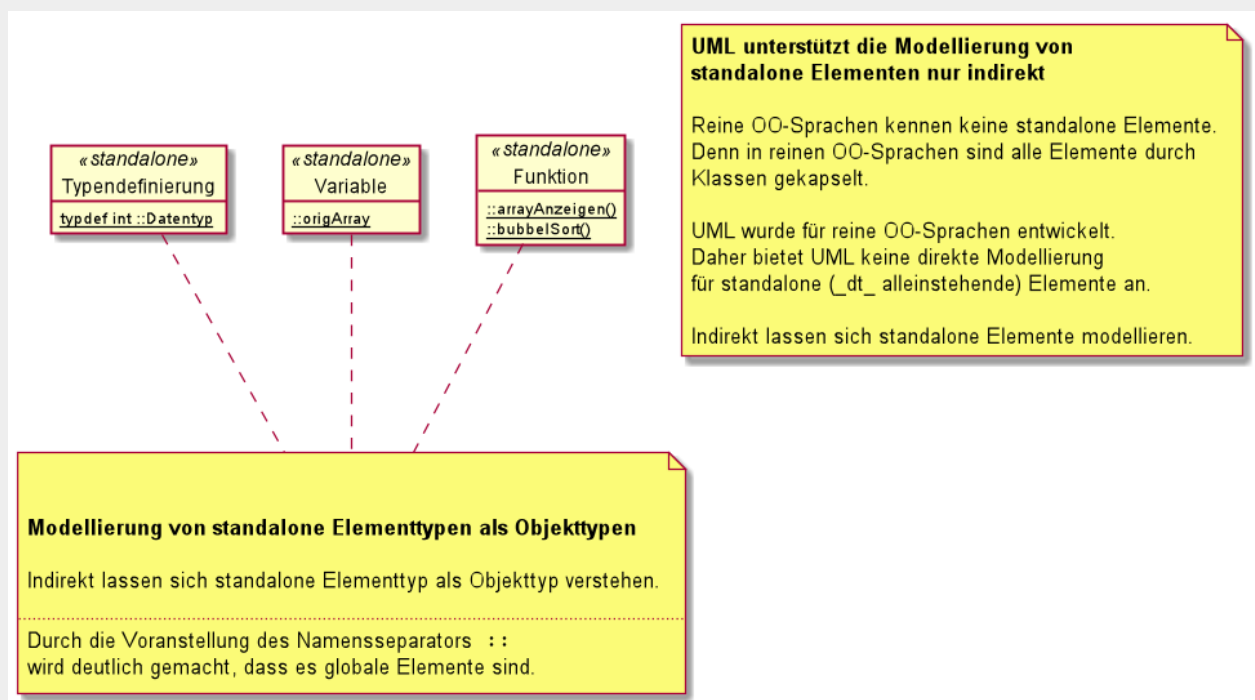


Figure 3. UML unterstützt die Modellierung von stand-alone Elementen nur indirekt.

Das zweiten Code-Lesen

Klären, wie vorhandene Code-Bruchstücke für die eigene Aufgabe wiederverwendet werden können.

Das erste Code-Lesen bot uns eine erste Orientierung: Was wird an Code angeboten, Wofür wurde geschrieben?

Jetzt beim zweiten Code-Lesen stehen eher *wie*-Fragen im Vordergrund: *Wie können wir den angeboten Code Nutzen für die aktuelle Aufgabe?*

Es ist klug, gleich eine zweite Frage zu stellen: *Wie können wir den angebotenen Code in keinen Schritten so umschreiben (engl. to refactor), dass der Code nicht nur der aktuellen Aufgabe dient, sondern auch jederzeit für eine zukünftige Aufgabe hilfreich ist?*

Damit ergibt sich eine weitere Regel für zweite Code-Lesen:

Klären, welche Code-Bruchstücke über die aktuelle Aufgabe hinaus nützlich sind.

Damit verbunden ist eine allgemeine Frage: *Wie können wir vorhandenen Code eine Qualität geben, dass seine Wiederverwendung in zukünftigen Projekten sehr wahrscheinlich ist?*

Ein Gedanke zum Thema Software und Qualität

Quality isn't getting a computer to do what you want efficiently —though that can be a challenge at times. No, writing code for computers to read is easy; writing code for humans to read is hard. The hard part isn't writing, it's reading. As code increases, reading complexity grows $O(2^n)$ — exponentially.

— James Fulford in a [blog](#)

Das Zitat vertritt die Meinung, dass die Qualität von Software davon abhängt, wie gut lesbar der Code für die Menschen ist, die ihnen weiterentwickeln wollen. So kommen wir zum Rüstzeug für diese Aufgabe.

Rüstzeug für das zweite Lesen

Das Rüstzeug für gute Lesbarkeit von Code kennen

Ein gut lesbarer Text hat eine gute inhaltliche Gliederung.
So braucht auch gut lesbare Software eine klare Gliederung.

Ein Mittel, um die Lesebarkeit von Code zu erhöhen, ist die Benutzung von Namensräumen (*engl. namespace*). Namensräume machen die Gliederung von Code transparent. Je mehr Code geschrieben wird, um so wichtiger wird für seine Qualität eine nachvollziehbare Gliederung.

Beispielhaft werden hier ein paar Gliederungsgedanken zum vorhandenen Code vorgestellt, denn gliedern erhöht die Lesbarkeit und damit die Qualität von Software. Das ist der Kern von dem Zitat von James Fulford:

Die Kernaussage von James Fulford zum Thema software und Qualität

Qualität ist nicht einen Computer dazu zu bringen, auf eine effiziente Weise das zu tun, was du möchtest. ... Das Schreiben von Code, der durch den Computer gelesen werden kann, ist der leichte Teil des Codierens.

Der schwierigere Teil ist, so zu codieren, dass jeder andere Entwickler sich leicht in den Code einlesen kann.

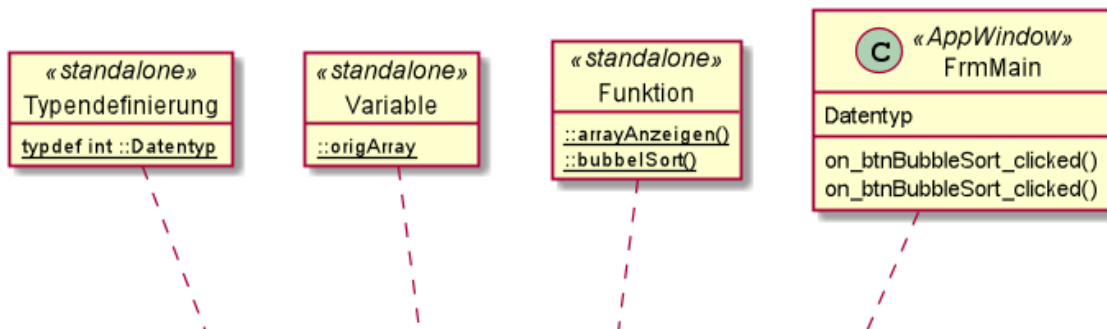
— James Fulford in a [blog](#)

Die klare und transparente Kapselung von Code ist ein Stilmittel, um Code zu gliedern und damit die Lesbarkeit des Codes zu erhöhen.

Ein Mittel zur Kapselung von Code sind **Namensräume** (*engl.* namespace).

Namensräume sind ein wichtiger Zwischenschritt, wenn Code nicht nur durch *Copy and Paste* wiederverwendet werden soll.

Deutlich macht dies auch eine Regel aus den *C++ Core Guidelines*, auf die wir im nächsten Abschnitt eingehen.



Klären, wie vorhandene Code-Bruchstücke für die eigene Aufgabe wiederverwendet werden können

Datentyp

Der Typ wird in der Signatur der Methoden `arrayAnzeigen` und `bubbelSort` benutzt. Außerdem wird im Body von `FrmMain::on_btnBubbleSort_clicked()` benutzt.

=> Die standalone Typendefinierung bleibt global.

origArray

wird lediglich im Body von `FrmMain::on_btnBubbleSort_clicked` benutzt, ohne zuvor als Parameter übergeben zu sein.

=> Die AppWindow Klasse `FrmMain` ist die top-level Klasse der Anwendung.

=> Statt `origArray` im globalen Namespace von C++ zu deklarieren verwenden wir `origArray` als Eigenschaft der Anwendung.

arrayAnzeigen()

Die standalone Funktion unterstützt das Anzeigen eines Array in einem ListWidget von Qt.

=> Denkbar wäre die standalone Funktion in einen Namensraum zu verschieben, der diese Aufgabe deutlich macht.

Denn diese Code-Fragment erfüllt eine allgemeine Aufgabe, die bestimmt auch in anderen Qt Applikationen auftritt.

Ein Namensraum `qtHelper` könnte solche allgemein wiederverwendbare Code-Fragmente sammeln.

Hier ein Vorschlag für einen Namensraum

```
qtHelper::listWidget::arrayAnzeigen()
```

bubbelSort()

Die standalone Funktion ist eine Implementierung eines Sortieralgorithmus.

=> Denkbar wäre die standalone Funktion in einen Namensraum zu verschieben, der diese Aufgabe deutlich macht.

Hier ein Vorschlag für einen Namensraum

```
algorithmus::sort::bubbelSort()
```


Rüstzeug für guten C++-Code

Rüstzeug für guten C++-Code: Die C++ *Core Guidelines* kennen

Die C++ *Core Guidelines* sind ein guter Ratgeber.

Ein Ratgeber ist immer dann hilfreich, wenn man einen Rat sucht.

Wer einen Ratgeber durchliest, ohne im Hinterkopf einen bestimmten Rat zu suchen, kann auch erleben, dass das Durchlesen eines Ratgebers recht verstörend sein kann.

Deshalb wollen wir erstmal klären, zu welchen Themen wir Rat suchen.

Welchen Rat geben die Core Guidelines zur *Code Organisation*?

Welchen Rat gibt es zu globalen Variablen in den C++ Core Guidelines?

Die [Regel I2 der Core Guidelines](#) lautet: *Vermeide nicht konstante globale Variablen.*

Die Regel gehört zur Hauptabschnitt [I: Interfaces](#). Als Erläuterung zu Interfaces (dt. Schnittstellen) heißt es in den Core-Guidelines:

Gute *Interfaces* zu haben, ist wohl der wichtigste einzelne Grundbaustein für eine gute Organisation des Codes. Gute Interfaces zeichnen sich dadurch aus, dass sie leicht zu verstehen sind, so gestaltet sind, dass sie ihrer effizienten Benutzung einfach ist und einladend, weil sie nicht fehleranfällig sind, die Schreiben von Tests unterstützen.

— [I: Interfaces](#)

Hilfestellung zur App-Entwicklung:

NOTE

- Welche nicht konstanten globalen Variablen enthalten die Code-Bruchstücke?
- Wie können diese Variablen deklariert werden, ohne *globale* Variablen zu sein?

Wie äußert sich die Core Guideline zur Nützlichkeit von Namensräumen?

In der [Regel NR.4](#) aus dem Unterabschnitt [NR: Non rules and myths](#) geht es darum, dass es nicht sinnvoll ist für jede Klasse eine Quelldatei (engl. source file) zu eröffnen. In der *Alternative* der Regel heißt es:

Benutze als Alternative *Namespaces*.

Denn Namespaces enthalten logisch zusammenhängende (engl. cohesive) Sätze von Klassen und Funktionen.

— [Regel NR.4: Do not insist on placing each class declaration in its own source file](#)

Hilfestellung zum Lesen der Core Guidelines

- Jede Regel gehört zu einem bestimmten Hauptabschnitt (*engl.* major section) der Guidelines oder zu einem Unterabschnitt (*engl.* subsection).

Jeder Hauptabschnitt und Unterabschnitt wird zum vereinfachten Referenzieren auch eine Buchstabenkombination zugeordnet. Diese Zuordnung wird in der [Regel In.sec der Core Guidelines](#) erläutert.

Das Referenzierungsnummer von Core Guideline Regel starten mit Buchstabenkombinationen

Die Referenznummer der [Regel In.sec](#)

macht deutlich: Die Regel gehört zum Abschnitt **In: Introduction**, der gemäß der [Regel In.sec](#) einer der Hauptabschnitte ist.

- Jeder Regel der Core Guidelines folgt einer festen Struktur.

Die Strukturelemente einer Regel beschreibt die [Regel In.struct](#) beschrieben. Die Strukturelemente, die eine Regel enthalten kann sind:

Die Strukturelemente einer C++ Core Guideline Regel

Jede Regel (Leitlinie, Vorschlag) setzt sich aus verschiedenen Abschnitten zusammen.

- Die Regel selbst
- Eine Referenznummer

Eine Referenznummer folgt diesem Aufbau:

<Kürzel für Hauptabschnitt>.<Regelnummer>

<Kürzel für Nebenabschnitt>.<Kürzel für Regelthema>

Since the major sections are not inherently ordered, we use letters as the first part of a rule reference “number”. We leave gaps in the numbering to minimize “disruption” when we add or remove rules.

— Regel **In.struct**

- *Argumente bzw. Gründe* für die Regel (*engl. reasons / rationales*) werden geben, damit die Regel für den Programmierer besser einsichtig ist.
- Um die Verständlichkeit einer Regel zu verbessern, werden *Beispiele* geben. Die Beispiel können negativ oder positiv Beispiele sein.
- Bei Regeln, die dazu aufrufen etwas zu vermeiden, werden *Alternativen* aufgezeigt.
- Regel müssen nicht an sich allgemeingültig sein, deshalb werden gegebenenfalls *Ausnahmen* (*engl. exceptions*) angeführt.
- Die *Durchsetzung einer Regel* (*engl. enforcement*) vereinfacht sich, wenn es Mechanismen gibt die ihre Umsetzung überprüfen.

Solche Überprüfungen können auf verschiedene Weise erreicht werden: Ein Code Reviewing, also durch eine Person; aber auch mechanistisch automatisiert durch: statische Analysen, den Compiler oder Laufzeitüberprüfungen. Mehr dazu in der **Regel In.force**

- *Verweise* (*engl. see alsos*) geben Hinweise auf verwandte Themen.
- *Anmerkungen* oder *Kommentare* (*engl. notes / comments*) enthalten eine Information, die sich den bisher genannten Elementen zur Klassifizierung der Information nicht zu ordnen ließen.
- Das Element *Diskussion* (*engl. discussion*) vertieft typischerweise die Gründe einer Regel und/oder gibt weitergehende Beispiele.

Anmerkung zur Typdefinierung: **Datentyp**

Die vorgegebenen Code-Bruchstücke geben dem Array, der durch den Sortieralgorithmus bearbeitet werden soll, den Typ **Datentyp**. Konkreter formuliert: Es wird nicht der Array selbst an die Funktion **bubbleSort** übergeben, sondern ein Pointer auf den Array.

Beim ersten Lesen hatten wir die folgenden Fragen im Hinterkopf:

- Was macht ein Code-Abschnitt?
- Wofür wird der Code-Abschnitt gebraucht?

Spätestens jetzt ist ein guter Zeitpunkt sich die folgenden Fragen zu beantworten:

Was macht die Typdefinition **Datentyp**?

Sie ist eine Art Tarnung, wodurch der eigentliche Datentyp **int** verschleiert wird.

Wofür wird diese Typdefinition **Datentyp** gebraucht?

Zunächst mal kann man dazu in den Code schauen:

- Die Typdefinierung wird benutzt, um eine globale Variable **origArray** zu deklarieren als **datentyp***.

Das heißt **origArray** ist ein Zeiger auf einen Speicheradresse vom Typ **datentyp**.

- Die deklarierte Variable **origArray** wird in den Code-Bruchstücken lediglich in einer Zuweisung in der Funktion **FrmMain::on_btnBubbleSort_clicked()** benutzt:

```
tmpArray = origArray;
```

- Die Typdefinierung wird benutzt als Parametertyp in zwei Funktionen:

```
void arrayAnzeigen(datentyp *array, int anz, QListWidget* lwAnzeige)
void bubbleSort(datentyp *feld, int anz) //aufsteigendes Sortieren
```

Hilfestellung zur Frage: Wofür wird die Typdefinierung **datentyp** gebraucht?

- Wie verändert sich der Code dieser Funktion, wenn statt der Typdefinierung

```
typedef int dentyp
```

die Typdefinierung

```
typedef double dentyp
```

verwendet wird?

Jetzt erhalten wir eine Antwort auf die Frage:

NOTE

Wofür wird der Typdefinierung **datentyp gebraucht?**

Mittels der Typdefinierung kann ein einfaches Beispiel für eine *generische Programmierung* gegeben werden.

Gewinne ein Verständnis zum Schlagwort *generische Programmierung*.

Zum Beispiel durch das Durchdenken unserer obigen Frage:

Wie verändert sich der Code dieser Funktion, wenn der Typ **datentyp** nicht den Typen **int**, sondern den Typ **double** repräsentiert.

oder über die folgenden Links:

- [Generische_Programmierung @ de.wikipedia.org](https://de.wikipedia.org/wiki/Generische_Programmierung)
- [Was ist generische Programmierung? @ berti-cmm.de](https://berti-cmm.de/2017/01/02/was-ist-generische-programmierung/)

- Die Typdefinierung wird benutzt zur Deklaration bzw Definition von Variablen in Funktionen benutzt.
 - Für eine Deklaration in der Funktion **bubbelSort**

```
dentyp *tmp;
```

Hilfestellung zur Fehlersuche in den Code-Bruchstücken

- Vergleich die Deklaration der Variable `tmp` mit ihrer Benutzung im Code der Funktion.

NOTE

```
void bubbleSort(datentyp *feld, int anz) //aufsteigendes
Sortieren
{
    datentyp *tmp;
    for (int x=0; x<anz-1; x++) {
        for (int i=0; i<=anz-1; i++) {
            if (feld[i]<feld[i+1]) {
                tmp = feld[i];
                feld[i+1] = feld[i];
                feld[i+1] = tmp;
            } //if
        } //for i
    } // for x
}
```

- Vergleiche den gezeigt Sortieralgorithmus mit seiner Kommentierung als *aufsteigendes Sortieren*.
- Für eine Definition in der Funktion `FrmMain::on_btnBubbleSort_clicked`

```
datentyp* tmpArray = new datentyp[anzElemente];
```

Hilfestellung zur App-Entwicklung

Hier tritt eine bisher unbekannte Variable auf `anzElemente`.

NOTE

- Was repräsentiert die Variable?
- Wofür wird die Variable gebraucht?
 - Wie bekommt die Variable einen Wert?
 - Wo wird die Variable deklariert?