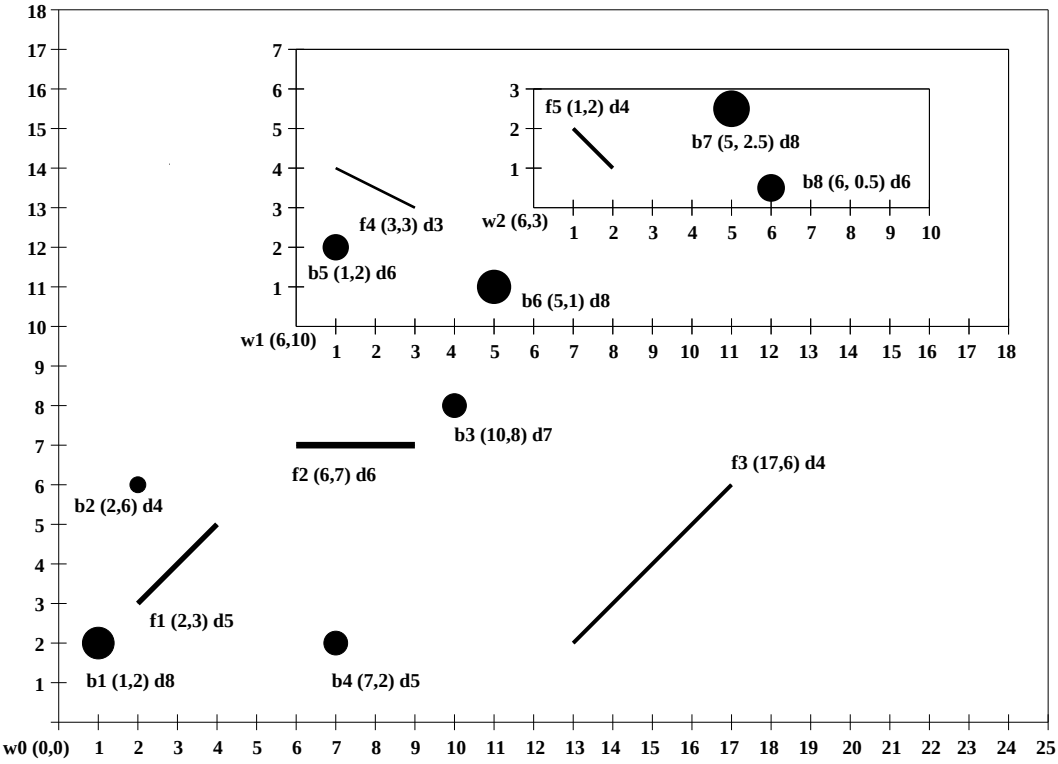
Praktikumsaufgabe 2

In diesem zweiten Praktikum lernen Sie zunächst das *Refactoring* von bestehendem Quellcode (der im ersten Praktikum erstellt wurde) kennen.
Das Refactoring bezeichnet die Strukturverbesserung von Quelltexten unter Beibehaltung des Programmverhaltens mit dem Ziel, die Lesbarkeit, Verständlichkeit, Wartbarkeit und die Erweiterbarkeit zu verbessern. Es ist ein zentraler Bestandteil der *Agilen Softwareentwicklung*.
Der Hauptgrund für das in diesem Praktikum durchgeführte Refactoring ist es, die leichte Erweiterbarkeit des Codes an neue Anforderungen zu verbessern.
Wie in der nachfolgenden Abbildung dargestellt, soll es nun auch möglich sein, neben Bohrungen und Fräsungen auch (kleinere) Werkstücke, (auf denen wiederum Bohrungen Fräsungen oder (kleinere) Werkstücke als Komponenten vorhanden sind) auf ein Eltern-Werkstück (`parent`) aufzubringen. Jedes kleinere aufgebrachte Werkstück besitzt einen Verankerungspunkt (die Koordinaten des linken unteren Eckpunktes) und ein eigenes lokales Koordinaten-System, welches verwendet wird um Komponenten auf diesem Werkstück abzulegen.
Z. B. ist das Werkstück `w1` eine Komponente von Werkstück `w0`. Das Attribut `parent` von `w1` zeigt also auf `w0`. Der Verankerungspunkt von `w1` bezüglich `w0` hat die Koordinaten $(6, 10)$. Entsprechend ist das Wektstück `w2` eine Komponente von Werkstück `w1`, besitzt also `w1` als `parent` und hat bezüglich `w1` den Verankerungspunkt $(6, 3)$.



Weiterhin werden Sie in diesem Praktikum die Verwendung des *Composite Entwurfsmusters* sowie das Konzept eines *Iterators* kennenlernen. Ein *Iterator* erlaubt es sequentielle Datenstrukturen (hier die doppelt verkettete Liste) zu durchlaufen.

I) Modellierung und Implementierung:

Vergessen Sie nicht **alle** Methoden, die das eigene Objekt nicht verändern, als `const` zu deklarieren (**const correctness**). Da dies eine sprachspezifische Eigenschaft von C++ ist,

wird sie im UML-Klassen-Diagramm nicht dargestellt.

1. **Abstrakte Klasse IKomponente:**

   Alle gemeinsamen (und später benötigten) Verhaltensweisen (Methoden) von Komponenten-Objekten (Bohrungen, Fräsungen, Werkstücke) werden in die abstrakte Klasse `IKomponente` ausgelagert. Diese Klasse wird die neue Wurzel der Vererbungshierarchie und dient als **Schnittstelle**. Die Vorsilbe `I` in ihrem Namen deutet an dass sie ein **Interface** ist.

   Sie besitzt keine Attribute und ihre Methoden sind **rein virtuell**, werden also nur deklariert und nicht definiert. In C++ wird dies durch ein nachfolgendes `= 0` bei der Deklaration kenntlich gemacht, Z.B.:

   ```
   virtual double calcTotalPath() const = 0;
   ```

   Wie aus den gewählten Namen der Methoden erkennbar ist, sind sie alle als `const` zu deklarieren, mit der einzigen Ausnahme der Methode
   `void setParent(const IKomponente* p)`.

   Da diese Klasse als Schnittstelle fungiert, verschiebt man die Überladung des Schiebeoperators als freie Funktion von der Header-Datei von `Komponente` in die Header-Datei von `IKomponente`, wobei nun als Typ für den zweiten Parameter `IKomponente` verwendet wird:

   ```
   inline std::ostream& operator<<(std::ostream& os, const IKomponente& ik) {
     ik.output(os);
     return os;
   }
   ```

   Dies erleichtert das Inkludieren.

   Es können keine Objekte von dieser abstrakten Klasse erstellt werden, sie besitzt deshalb auch keinen Konstruktor.

   Ihr virtueller Destruktor ist die einzige definierte Methode dieser Klasse. Die leere Implementierung

   ```
   IKomponente::~IKomponente() { }
   ```

   wird man in die zugehörige Datei `ikomponente.cpp` auslagern, damit eine virtuelle Methodentabelle (*VMT*) erzeugt wird.

   Unterklassen von `IKomponente` erben also keine Implementierung von Methoden sondern nur die Schnittstelle (Signatur) der Methoden. Man spricht in diesem Zusammenhang deshalb auch von **Schnittstellenvererbung** (Vererbung des Typs ohne Implementierung - engl. *Subtyping*) im Gegensatz zur **Implementierungsvererbung** (Vererbung von Typ und Implementierung - engl. *Subclassing*).

   Die Bedeutung der einzelnen Methoden wird später bei ihrer konkreten Implementierung in den Unterklassen genau erklärt.

2. **Struktur IKomponentenElement:**

   Aus der ursprünglichen Struktur `KomponentenElement`, wird nun die Struktur `IKomponentenElement`, die nun verwendet wird, um eine doppelt verkettete Liste von Komponentenobjekten zu erstellen (s. u.).

IKomponente
+calcTotalPath() : double
+getXAbsolute() : double
+getYAbsolute() : double
+getX() : double
+getY() : double
+distance(k : IKomponente const*) : double
+getParent() : IKomponente const*
+setParent(p : IKomponente const*) : void
+output(os : ostream&) : void
+~IKomponente()
Komponente
-x : double
-y : double
-parent : IKomponente const*
+Komponente(xPos : double, yPos : double)
+calcTotalPath() : double
+getX() : double
+getY() : double
+getXAbsolute() : double
+getYAbsolute() : double
+distance(k : IKomponente const*) : double
+getParent() : IKomponente const*
+setParent(p : IKomponente const*) : void
+output(ostream&) : void
-parent
<<Struct>>
IKomponentenElement
+k : IKomponente*
+before : IKomponentenElement*
+next : IKomponentenElement*
+next
+before
-curr
Iterator
-curr : IKomponentenElement*
+Iterator(begin : IKomponentenElement* = nullptr)
operator ++() : Iterator&
operator !=(it : Iterator const&) : bool
operator ==(it : Iterator const&) : bool
operator *() : IKomponente*
+~Iterator()
DeList
-counter : int
-first : IKomponentenElement*
-getElement(it : Iterator) : IKomponentenElement*
+DeList()
+DeList(rhs : DeList const&)
operator =(rhs : DeList const&) : DeList&
+~DeList()
+clear() : void
+size() : int
+erase(pos : Iterator) : Iterator
+push_back(k : IKomponente*) : void
+end() : Iterator
+begin() : Iterator
-komponenten
Werkstueck
-height : double
-width : double
-pathIsOptimized : bool
-komponenten : DeList
-getNearest(pk : IKomponente const*) : Iterator
+Werkstueck(xPos : double = 0.0, yPos : double = 0.0, h : double = 1, w : double = 1)
+getHeight() : double
+getWidth() : double
+add(k : IKomponente*) : void
+remove(k : IKomponente const*) : void
+getChild(i : int) : IKomponente const*
+calcTotalPath() : double
+calcPathLength() : double
+optimizePath() : void
+output(os : ostream&) : void
+~Werkstueck()
Bohrung
-diameter : double
+Bohrung(x_ : double = 0.0, y_ : double = 0.0, diam : double = 6.0)
+~Bohrung()
+getDiameter() : double
+output(os : ostream&) : void
Fraesung
-length : double
-angle : double
+Fraesung(x : double, y : double, diam : double, l : double, a : double = 0.0)
+calcTotalPath() : double
+getLength() : double
+getAngle() : double
+output(o : ostream&) : void
+getEndX() : double
+getEndY() : double

3. **Klasse Komponente:**

   Die Klasse `Komponente` erbt nun von der abstrakten Klasse `IKomponente`.

   Die Klasse `Komponente` erhält das zusätzliche Attribut `parent`, in der ein Verweis auf das Werkstück, dessen Bestandteil diese Komponente ist, gespeichert wird. Für dieses neue Attribut sind die zugehörigen Getter/Setter-Methoden zu implementieren.

   Die neue Methode `calcTotalPath()` liefert einfach den Ergebniswert `0.0` zurück.

   Die Methoden `getX()` und `getY()` bleiben unverändert, denn ihre zugehörigen Attribute `x, y` sind immer noch die Koordinaten der Komponente bezüglich des lokalen Koordinaten-Systems des `parent`-Werkstückes.

   Die Methoden `getXAbsolute()` bzw. `getYAbsolute()` liefern die absoluten Koordinaten bezüglich des alles umfassenden Werkstückes (im Beispiel der Abbildung sind dies genau die Koordinaten bezüglich des Werkstückes `w0`).
   Tipp: Bei der Implementierung hilft die Polymorphie. Wenn `parent != nullptr` reicht es zu den lokalen Koordinaten `x` bzw `y` den Wert `parent->getXAbsolute()` bzw. `parent->getYAbsolute()` zu addieren (warum?), andernfalls können sofort `x` bzw `y` zurück gegeben werden.

   Die Methode `distance(const IKomponente* k)` liefert den euklidischen Abstand von `*this` zu `*k` zurück. Günstigerweise verwendet man für die Berechnung des euklidischen Abstandes die absoluten Koordinaten und vermeidet so eine Fallunterscheidung wenn eine der Komponenten ein Werkstück ist.

   Die Methode `output(std::ostream & os)` bleibt unverändert.

4. **Klasse Bohrung:**

   In der Klasse `Bohrung` sind keine Änderungen notwendig.

5. **Klasse Fraesung:**

   Die einzige Änderung, die in der Klasse `Fraesung` notwendig ist, ist das Überschreiben der Methode `calcTotalPath()`. Um eine Fräsung sauber auszufräsen bewegt sich der Bohrkopf mit dem im Durchmesser passenden Fräsbohrer abgesenkt vom Startpunkt zum Endpunkt und kehrt dann im abgesenkten Zustand zurück zum Startpunkt um auf dem Rückweg die Fräsung zu glätten. Sein zurückgelegter Weg beträgt deshalb `2 * length`. Dieser Wert ist in der Methode `calcTotalPath()` zurück zu geben.

6. **Klasse Iterator:**

   Objekte vom Typ `Iterator` dienen hier zum Durchlaufen der doppelt verketteten IKomponenten-Liste mithilfe eines `DeList`-Objekts.

   Der Konstruktor dieser Klasse soll als nicht-konvertierender Konstruktor definiert werden.

   Jeder Iterator speichert seine aktuelle (current) Position in der Liste über das Attribut `curr`, das entweder den Wert `nullptr` hat oder auf ein bestimmtes Element vom Typ `IKomponentenElement` der Komponentenliste zeigt.

   Der Anfangswert dieses Attributes `curr` wird über den Konstruktor gesetzt.

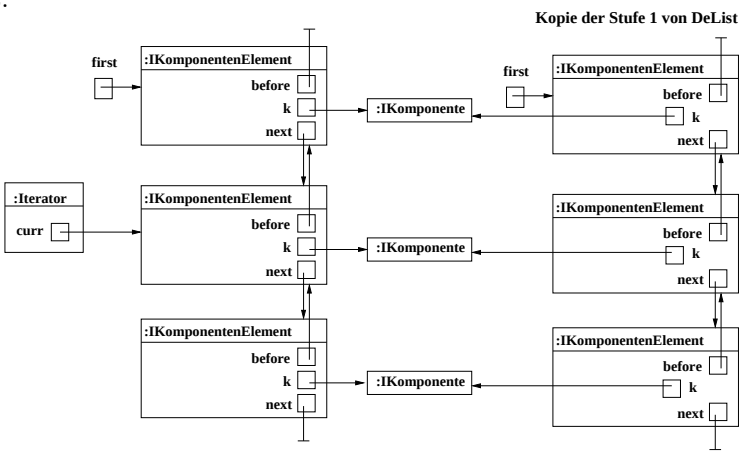
   Durch das Überladen der Präfix-Form des `++` Operators kann dieser Zeiger auf das Nachfolgende Element fortbewegt werden (*Tipp:* `curr = curr->next;`). Dabei erhält dieser Zeiger automatisch den Wert `nullptr` wenn die Ausgangsposition keinen Nachfolger besitzt.

   Das Überladen der beiden Vergleichsoperatoren ermöglicht den Adressvergleich mit einer anderen Iteratorposition.

Das Überladen des Dereferenzierungsoperators liefert den Zeiger `curr->k` vom Typ `IKomponente*` zurück.

7. **Klasse DeList:**

   Die Klasse `DeList` geht durch größere Änderungen aus der Klasse `KomponentenList` hervor und ersetzt diese. Die Funktionalität der Methoden bleibt erhalten aber für Positionsangaben wird ein Iterator-Objekt Anstelle eines Integer-Wertes (Index) verwendet.



   Das Attribut `counter` speichert die Anzahl der in der Liste aktuell verketteten Objekte des Typs `IKomponentenElement`. Dies ist auch der Rückgabewert der Methode `size()`.

   Das Attribut `first` ist ein Zeiger auf das erste in der Liste gespeicherte Element. Im Fall einer leeren Liste hat `first` den Wert `nullptr`.

   Die private Hilfsmethode `getElement(Iterator it)` liefert einen Zeiger auf das `IKomponentenElement` auf das der Iterator `it` zeigt. Sollte der übergebene Iterator auf kein Element der eigenen Liste zeigen, ist `nullptr` zurück zu geben.

   Diese Hilfsmethode wird nur noch zur Implementierung der Methode `erase()` verwendet.

   Die Methode `erase(Iterator pos)` löscht das Element aus der Liste, auf das der Iterator `pos` zeigt. Wenn dieses Element existiert, wird ein Iteratorobjekt zurück gegeben, das auf den Nachfolger zeigt. Ansonsten wird ein Iteratorobjekt, dessen Attribut `curr` den Wert `nullptr` hat, zurück gegeben.

   Die Methode `at()` der Klasse `KomponentenList` entfällt nun, da ihre Funktionalität über den Dereferenzierungsoperator des Iterators umgesetzt wird.

   Die Methode `begin()` liefert nun ein Iteratorobjekt zurück, dessen Attribut `curr` den Wert `first` hat.

   Die Methode `end()` liefert nun ein Iteratorobjekt zurück, dessen Attribut `curr` immer den Wert `nullptr` hat.

   Achtung: Bei der Verwendung eines Iteratorobjekts im Zusammenspiel mit einem `DeList`-Objekts soll anstelle des Werts `nullptr` nur noch die Methode `end()` verwendet werden.

   Die Methode `push_back(IKomponente* k)` fügt das über den Zeiger `k` übergebene Komponentenobjekt, verpackt in der Struktur `IKomponentenElement` (welches mit `new` anzulegen ist), am Ende der Liste ein. Um einen Zeiger auf das letzte Element der Liste zu erhalten kann nun nicht mehr die Hilfsmethode `getElement()` verwendet werden. Stattdessen muss man jetzt selber einen Zeiger beginnend bei `first` bis an das Ende fortbewegen.

Da das Löschen aller in der Liste gespeicherten Elemente nicht nur im Destruktor, sondern auch im Copy-Zuweisungsoperator benötigt wird, wird diese Funktionalität in die neue Methode `clear()` ausgelagert. Dabei kann man ähnlich wie beim Destruktor der Klasse `KomponentenList` vorgehen.

Im Destruktor der Klasse `DeList` muss dann nur noch die Methode `clear()` aufgerufen werden.

Im Copy-Konstruktor `DeList(const DeList& rhs)` muß **keine exakte tiefe Kopie** der Liste erzeugt werden, sondern es reicht eine **Kopie der Stufe 1** (siehe obige Abbildung). D.h. es wird eine doppelt verkettete Liste mit eigenen Elementen vom Typ `IKomponentenElement` angelegt. Die einzelnen Zeiger `k` eines IKomponentenElements zeigen aber auf dasselbe `IKomponenten`-Objekt, auf das auch das `IKomponentenElement`-Objekt von `rhs` zeigt. Somit müssen keine Kopien der `IKomponenten`-Objekte erzeugt werden.

Im Copy-Zuweisungsoperator `operator=(const DeList& rhs)` reicht ebenfalls eine **Kopie der Stufe 1**, wenn keine Selbstzuweisung vorliegt. Zum Löschen der alten Liste kann dann wiederum die Methode `clear()` verwendet werden.

8. **Klasse Werkstueck:**

   Um Objekte zu Baumstrukturen hinzufügen und um Teil-Ganzes-Hierarchien zu realisieren, eignet sich die Verwendung des Entwurfsmusters Composite/Kompositum (vgl. Skript oder ausführlicher das bekannte Buch der Gang of Four Gamma, Helm, Johnson, Vlissides).

   Es ermöglicht zudem einzelne Objekte (hier Bohrungen, Fräsungen) als auch Kompositionen von Objekten (hier `Werkstueck`-Objekte) einheitlich zu behandeln (durch Verwendung der Methoden `output()` und `calcTotalPath()`, die die Rolle, der im Skript aufgeführten Methode `operation()` übernehmen).

   Bis auf die Klasse `Werkstueck`, die die Rolle von `Composite` übernimmt, sind für die Anwendung dieses Entwurfsmusters alle Klassen vorbereitet: Die Klasse `IKomponente` übernimmt die Rolle von `Component`, die Klasse `Bohrung` übernimmt die Rolle von `Leaf` bzw. `SimpleComponent`.

   Sinnvoll ist hier eine sehr häufig eingesetzte Variante dieses Entwurfsmusters, bei der die Deklaration der Methoden für die Verwaltungsoperationen von Kindobjekten (`add()`, `remove()`, `getChild()`) nur in der Composite-Klasse erfolgt.

   Zusätzlich zu den Koordinaten des Verankerungspunktes und einem Verweis auf seine Elternkomponente (`parent`) besitzt ein Werkstück noch die Attribute Höhe (`height`), Breite (`width`), das boolsche Flag `pathIsOptimized`, welches bei der Optimierung der Pfadlänge des Bohrkopfes (s. u.) zum Einsatz kommt, und eine doppelt verkettete Liste `komponenten` in der die Kindkomponenten abgespeichert werden.

   Im Konstruktor wird das Attribut `pathIsOptimized` auf den Wert `false` gesetzt.

   Die Getter-Methoden `getHeight()` und `getWidth()` liefern den Wert des jeweiligen Attributes zurück.

   Die Methode `add(IKomponente* k)` fügt die als Zeiger übergebene Komponente am Ende der Komponenten-Liste `komponenten` ein nachdem `this` als `parent` dieser Komponente gesetzt wurde.

   Die Methode `remove(const IKomponente* k)` löscht die Komponente, auf die der übergebene Zeiger `k` zeigt aus der Komponenten-Liste `komponenten` und setzt `parent` der Komponente zurück auf `nullptr`.

   Die Methode `getChild(int i)` liefert einen Zeiger auf die Komponente, die unter Index-Position `i` in der Komponenten-Liste gespeichert ist, zurück. Wie üblich beginnt die Index-Nummerierung bei 0.

Die Methode `calcPathLength()` berechnet nur die reine Länge des Weges, die der Bohrkopf zurück legt, um vom eigenen Verankerungspunkt die Koordinaten der Kinder anzufahren und dann wieder zu seinem Verankerungspunkt zurück zu kehren.

Z. B. wenn im Werkstück `w1` der obigen Abbildung die Kinder in der Reihenfolge `b5`, `b6`, `f4`, `w2` abgespeichert sind, berechnet diese Methode die Länge des Weges wie folgt:

In lokalen Koordinaten bzgl. des Werkstückes `w1`

```
(0,0) -> (1, 2) -> (5, 1) -> (3,3) -> (6,3) -> (0,0)
```

In absoluten Koordinaten wird dieser Weg beschrieben durch

```
(6,10) -> (7, 12) -> (11, 11) -> (9,13) -> (12,13) -> (6,10)
```

und er hat die Länge 18.89580.

Der zurückgelegte Weg des Bohrkopfes für das Ausfräsen der Fräsung `f4` und der zurückgelegte Weg um die Kinder von `w2` zu bearbeiten wird von dieser Methode also nicht berücksichtigt, dies geschieht erst in der Methode `calcTotalPath()` (s.u.).

Tipp: Bei der Implementierung wird man die geerbte Methode `distance()` verwenden.

Die Methode `calcTotalPath()` berechnet den total vom Bohrkopf zurückgelegten Weg, um alle Komponenten des Werkstückes zu bearbeiten.

Hierfür wird man am Günstigsten zunächst für alle Kinder wiederum die Methode `calcTotalPath()` aufrufen und die Ergebnisse aufsummieren. Zu dieser Summe wird man dann noch das Ergebnis der eigenen Methode `calcPathLength()` addieren.

Die Methode `output(ostream& os)` soll ein Werkstück wie im unteren
Abschnitt II) Test der Implementierung gezeigt, ausgeben.

Man beachte, dass entsprechend der Verschachtelungstiefe des Werkstückes alle seine Ausgaben einzurücken sind. Z. B. befindet sich Werkstück 1 auf der 1-ten Verschachtelungstiefe. Deshalb sind alle Ausgaben, die zu diesem Werkstück gehören um 1 Leerzeichen eingerückt. Entsprechend werden alle Ausgaben, die zum Werkstück `w2` gehören um 2 Leerzeichen eingerückt.

Die Verschachtelungstiefe läßt sich über den `parent` Zeiger ermitteln, indem man sich mit `parent = parent->getParent();` solange nach oben hangelt und mitzählt, bis `parent == nullptr` ist, also das Wurzel-Werkstück erreicht wurde.

Tipp: Anstelle des Mitzählens kann ein `std::string`-Objekt namens `blanks` jeweils um ein Leerzeichen ergänzt werden.

Für die Ausgabe der Komponenten kann man dann wie folgt vorgehen (dies zeigt dann auch wie man einen Iterator verwendet):

```
Iterator it = komponenten.begin();
while (it != komponenten.end()) {
  if (dynamic_cast<Werkstueck*>(*it) == nullptr) {
    os << blanks;
  }
  os << *(*it);
  if (dynamic_cast<Werkstueck*>(*it) == nullptr) {
    os << endl;
  }
  ++it;
}
```

Die private Hilfsmethode `getNearest(const IKomponente* pk)` sucht unter den Elementen in der Komponenten-Liste das Element aus, welches den kleinsten euklidischen Abstand zur Komponente hat, auf die der Übergabeparameter `pk` zeigt. Die Rückgabe erfolgt in der Form eines Iterators, dessen `curr` Attribut auf das entsprechende `IKomponentenElement` zeigt. Man kann dabei davon ausgehen, dass das `IKomponente`-Objekt, auf das `pk` zeigt, nicht selber in der Komponenten-Liste vorhanden ist, denn im Algorithums, der in der Methode `optimizePath()` implementiert ist (s. u.), wird genau dieses Objekt vorher aus der Komponenten-Liste entfernt.

Die Methode `optimizePath()` löst das Rundreise-Problem (Traveling Sales Man Problem (TSP)) approximativ, indem beginnend vom Verankerungspunkt des Werkstückes eine nächste Nachbarsortierung seiner Komponenten bestimmt wird.

Bis heute ist kein Algorithmus bekannt, der in polynomialer Laufzeit das TSP-Problem exakt löst. Hier sind schlaue, junge, helle Köpfe gefragt!!

Aufgrund der Vermutung $P \neq NP$, die ein Algorithmus mit polynomialer Laufzeit für das TSP sofort widerlegen würde, wäre man dann auch ganz sicher ein Kandidat für den Turing-Award oder die Fields-Medaille.

Ein möglicher exakter Lösungsalgorithmus besteht darin, von allen Permutationen der Komponenten-Sortierung die Pfadlänge zu bestimmen und daraus die Sortierung mit der minimalen Pfadlänge abzuleiten. Bezeichnet n die Anzahl der Komponenten so wächst die Laufzeit von diesem Lösungsalgorithmus wie $O(n!)$, also exponentiell!

Um Sie nicht zu überfordern geben wir im folgenden eine mögliche Implementierung der Bestimmung der nächsten Nachbarsortierung der Komponenten an.

Profis dürfen sich natürlich auch eine eigene Implementierung vornehmen!

```
// Solve approximately the Traveling Sales Man Problem (TSP) using a
// nearses neighbour strategy:
// round trip starts at the linking point of the Werkstueck.
// then go always to the nearest neighbour among the components
// that are not already on the new path
void Werkstueck::optimizePath() {
  if (!pathIsOptimized && komponenten.size() > 0) {
    // first call optimizePath for all Components,
    // which are of type Werkstueck recursively
    Iterator it = komponenten.begin();
    while (it != komponenten.end()) {
      Werkstueck* w = dynamic_cast<Werkstueck*>(*it);
      if (w != nullptr) {
        w->optimizePath();
      }
      ++it;
    }
    // Now determine a nearest neighbour-sorting
    // of all components
    DeList komponentenSortiert;
    IKomponente* pcurrent = this;
    Iterator nearest = getNearest(pcurrent);
    komponentenSortiert.push_back(*nearest);
    pcurrent = *nearest;
    komponenten.erase(nearest);
    while (komponenten.size() > 0) {
      nearest = getNearest(pcurrent);
      komponentenSortiert.push_back(*nearest);
      pcurrent = *nearest;
      komponenten.erase(nearest);
```

```
    }
    komponenten = komponentenSortiert;
    pathIsOptimized = true;
  }
}
```

Frage: In welcher/welchen Methode(n) der Klasse `Werkstueck` ist das Attribut `pathIsOptimized` wieder auf `false` zu setzen?

II) Test der Implementierung:

1. **Einfacher Test in einer Hauptfunktion:**

   Man erstelle ein Hauptprogramm `P02Main.cpp`, in dem alle (s. obige Abbildung) dargestellten Objekte vom Typ `Bohrung`, `Fraesung` und `Werkstueck` erstellt werden.

   Dabei sollen bei einem Werkstück **immer zuerst alle Bohrungen, dann alle Fräsungen und zum Schluss alle Teil-Werkstücke mit aufsteigender Nummerierung** hinzugefügt werden.

   Man gebe dann mittels Schiebeoperator das Werkstück `w0` aus.

   Weiterhin gebe man für alle Werkstücke die Ergebnisse aus, die die Aufrufe der Methoden `calcTotalPath()` und `calcPathLength()` liefern.

   Nun rufe man für das Werkstück `w0` die Methode `optimizePath()` auf und wiederhole alle obigen Ausgaben.

   Man sollte dann folgende Ergebnisse erhalten:

```
Werkstueck:
Verankerung: (0, 0)
height: 18, width: 25
Komponenten:
Bohrung: (1, 2), Durchmesser: 8
Bohrung: (2, 6), Durchmesser: 4
Bohrung: (10, 8), Durchmesser: 7
Bohrung: (7, 2), Durchmesser: 5
Fraesung mit Start: (2, 3) und Endpunkt: (4, 5), Durchmesser: 5
Fraesung mit Start: (6, 7) und Endpunkt: (9, 7), Durchmesser: 6
Fraesung mit Start: (17, 6) und Endpunkt: (13, 2), Durchmesser: 4
 Werkstueck:
 Verankerung: (6, 10)
 height: 7, width: 18
 Komponenten:
 Bohrung: (1, 2), Durchmesser: 6
 Bohrung: (5, 1), Durchmesser: 8
 Fraesung mit Start: (3, 3) und Endpunkt: (1.00005, 4.00009), Durchmesser: 3
  Werkstueck:
  Verankerung: (6, 3)
  height: 3, width: 10
  Komponenten:
  Bohrung: (5, 2.5), Durchmesser: 8
  Bohrung: (6, 0.5), Durchmesser: 6
  Fraesung mit Start: (1, 2) und Endpunkt: (2, 1), Durchmesser: 4

total Path w0: 130.931
Path length w0: 66.4814
total Path w1: 41.4788
Path length w1: 18.8958
total Path w2: 18.1109
```

```
Path length w2: 15.2825

Optimierung der Pfad-Laenge:
Werkstueck:
Verankerung: (0, 0)
height: 18, width: 25
Komponenten:
Bohrung: (1, 2), Durchmesser: 8
Fraesung mit Start: (2, 3) und Endpunkt: (4, 5), Durchmesser: 5
Bohrung: (2, 6), Durchmesser: 4
Fraesung mit Start: (6, 7) und Endpunkt: (9, 7), Durchmesser: 6
 Werkstueck:
 Verankerung: (6, 10)
 height: 7, width: 18
 Komponenten:
 Bohrung: (1, 2), Durchmesser: 6
 Fraesung mit Start: (3, 3) und Endpunkt: (1.00005, 4.00009), Durchmesser: 3
 Bohrung: (5, 1), Durchmesser: 8
  Werkstueck:
  Verankerung: (6, 3)
  height: 3, width: 10
  Komponenten:
  Fraesung mit Start: (1, 2) und Endpunkt: (2, 1), Durchmesser: 4
  Bohrung: (5, 2.5), Durchmesser: 8
  Bohrung: (6, 0.5), Durchmesser: 6
Bohrung: (10, 8), Durchmesser: 7
Bohrung: (7, 2), Durchmesser: 5
Fraesung mit Start: (17, 6) und Endpunkt: (13, 2), Durchmesser: 4

total Path w0: 114.792
Path length w0: 53.7518
total Path w1: 38.0695
Path length w1: 16.2448
total Path w2: 17.3525
Path length w2: 14.5241
```

2. **Test mittels Google-Test-Framework:**

   Wer noch nicht erschöpft ist (*nur freiwillig*) implementiere in der Anlehnung an das Praktikum 1 mit der Hilfe des Google-Test-Frameworks Test für alle verwendeten Klassen.