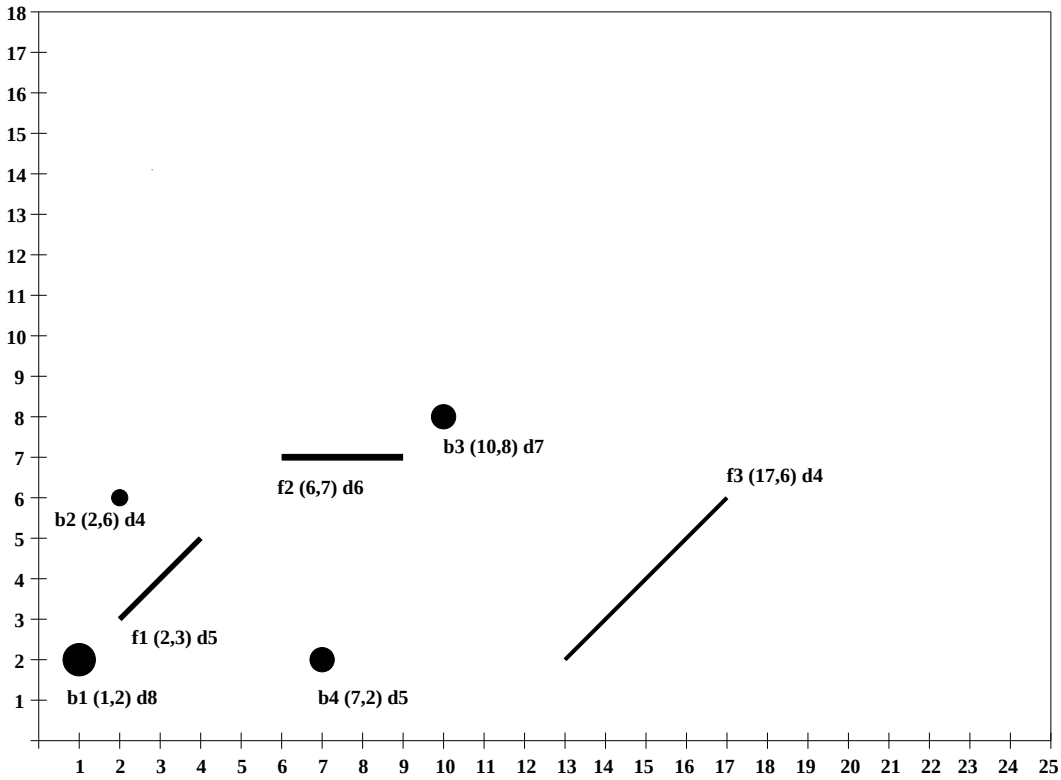
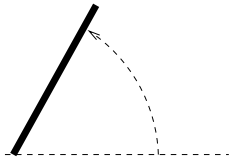
Praktikumsaufgabe 1

In den Praktika zur Vertieften Programmierpraxis soll die Länge des Weges eines Bohrkopfes, der auf einem Werkstück Bohrungen und Fräsungen aufbringt, *halbwegs* optimiert werden. Im letzten Praktikum wird dieser Weg dann auch mittels Qt visualisiert.
Im Verlauf aller Praktika werden Sie auch das Testen von Software (mittels Google-Test-Framework), das Auffinden von Memory-Leaks (valgrind, heob), den Umgang mit einem Debugger, die Verwendung von Design-Pattern und die Einarbeitung und Verwendung von großen Bibliotheken und deren API (Application Programmers Interface) kennen lernen.
Im ersten Praktikum fangen wir ganz einfach an und betrachten das in der folgenden Abbildung gezeigte Werkstück.



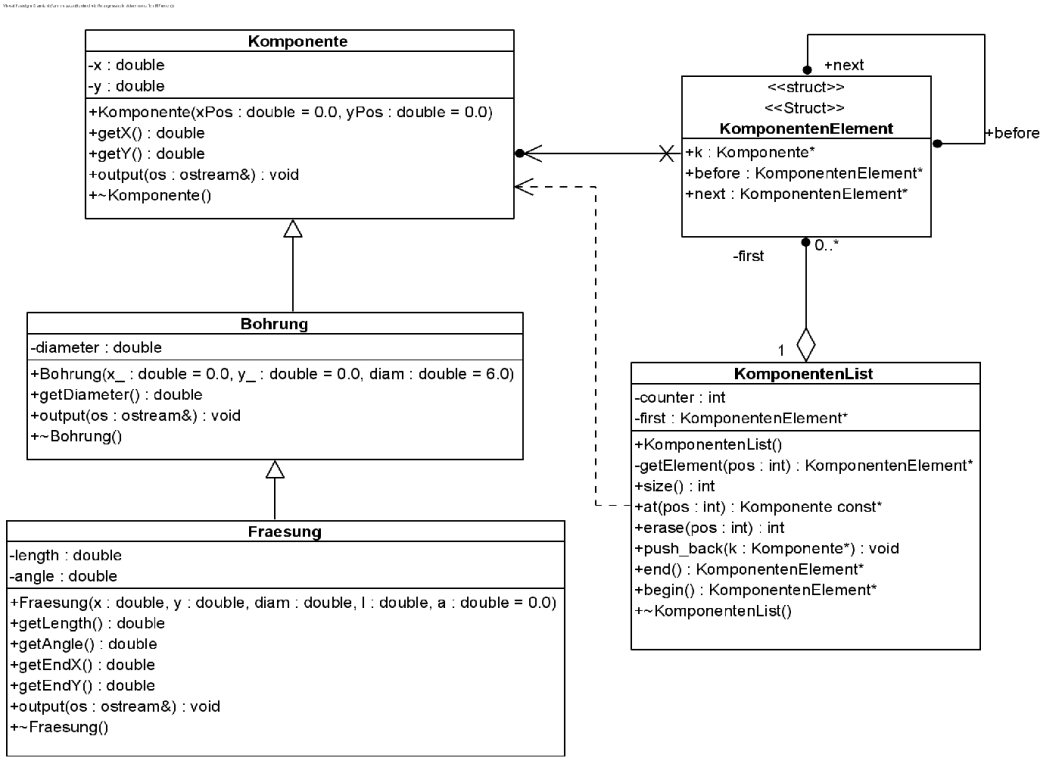
Der obere rechte Teil des Werkstückes wird erst im zweiten Praktikum weiter belegt werden. In gespannter Erwartung dürfen Sie sich aber schon jetzt auf diese zukünftige Erweiterung freuen.
Auf dem Werkstück sind Bohrungen und Fräsungen aufgebracht.
Bei einer Bohrung, z. B. b2, bezeichnet $(2, 6)$ die Koordinate des Mittelpunktes des Bohrloches in cm. Die Bezeichnung d4 gibt den Durchmesser des Bohrloches in mm an.



Bei einer Fräsung, z.B. f1, bezeichnet $(2, 3)$ die Koordinate des **Startpunktes** der Fräsung in cm. Die Bezeichnung d5 gibt den Durchmesser des Fräsbohrers in mm an. Bei einer Fräsung muss zusätzlich ihre Länge in cm und ihr Winkel (angle) **am Startpunkt** zur Horizontalen im Bogenmaß angegeben werden.

I) Modellierung und Implementierung:

Zur objektorientierten Modellierung der Bohrungen und Fräsungen eines Werkstückes bietet sich zunächst eine Vererbungshierarchie, dargestellt im unteren UML-Klassendiagramm, an. Alle gemeinsamen Eigenschaften (Attribute) und Verhaltensweisen (Methoden) werden in eine Basisklasse `Komponente` verschoben. Dies ermöglicht dann auch Polymorpie: Da sowohl Objekte vom Typ `Bohrung` als auch Objekte vom Typ `Fraesung` auch eine Instanz vom Typ `Komponente` sind, kann ein Zeiger vom Typ `Komponente*` auf sie verweisen. Ein derartiger Zeiger kann dann auch in einem **dynamisch allozierten** Objekt vom Typ `KomponentenElement` gespeichert werden. Somit können beide Arten von Objekten in einer `KomponentenList`, die Objekte vom Typ `KomponentenElement` doppelt verkettet, gespeichert werden.
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. **Klasse Komponente:**

   Man implementiere zunächst die Klasse `Komponente`. Hierbei beachte man die Default-Werte für die Parameter des Konstruktors.

   Die Methode `output(std::ostream& os)` schreibt die Koordinaten der Komponente eingerahmt durch ein rundes Klammerpaar in den übergebenen Stream `os`.

   In der Header-Datei von `Komponente` sollte man, wie in der Vorlesung gezeigt, auch den Schiebeoperator als freie Funktion überladen:

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

2. **Klasse Bohrung:**

   Man implementiere nun die Klasse `Bohrung`.

   Die Methode `output(std::ostream& os)` ist so zu überschreiben, dass ein Objekt vom Typ `Bohrung` in der unten angegebenen Form ausgegeben wird. Dabei kann man, die aus der Oberklasse (Elternklasse) `Komponente` geerbte Methode günstigerweise mit verwenden.

3. **Klasse Fraesung:**

   Man implementiere nun die Klasse `Fraesung`.

   Sollte für die Länge der Fräsung ein Wert kleiner 0 übergeben werden, so ist das Attribut `length` auf 0 zu setzen.

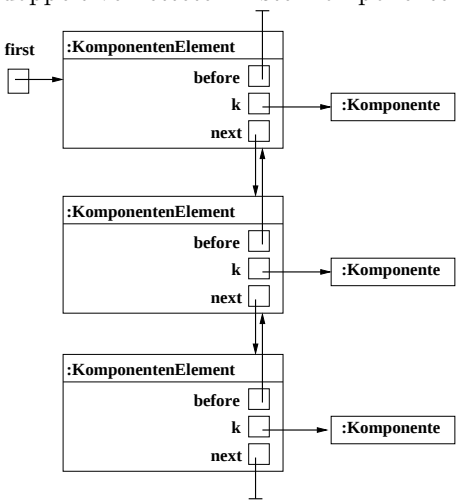
   Die Methoden `double getEndX()` und `double getEndY()` berechnen aus den Koordinaten des Startpunktes, der Länge und dem Winkel die Koordinate des Endpunktes und geben diese zurück.

   Die Methode `output(std::ostream& os)` ist so zu überschreiben, dass ein Objekt vom Typ `Fraesung` in der unten angegebenen Form ausgegeben wird. Dabei kann man wiederum die aus der Oberklasse `Komponente` geerbte Methode günstigerweise mit verwenden.

4. **Struktur KomponentenElement:**

   Man implementiere nun die Struktur `KomponentenElement`, die verwendet wird um eine doppelt verkettete Liste von Komponentenobjekten zu erstellen (s. u.).

5. **Klasse KomponentenList:** Man erstelle nun die Klasse `KomponentenList`, die es ermöglicht, in einer doppelt verketteten Liste Komponentenobjekte zu speichern.



   Das Attribut `counter` speichert die Anzahl, der in der Liste aktuell verketteten Objekte des Typs `KomponentenElement`. Dies ist auch der Rückgabewert der Methoden `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(int pos)` liefert einen Zeiger auf das `KomponentenElement` mit Index `pos`. **Wie bei Feldern üblich, hat auch hier das erste in der Liste gespeicherte Element den Index 0.** Sie hilft bei der Implementierung der Methoden `at()` und `erase()`.

   Die Methode `at(int pos)` liefert einen Zeiger auf die `Komponente` (Vorsicht: nicht `KomponentenElement`) die im `KomponentenElement` mit Index `pos` gespeichert ist.

Die Methode `erase(int pos)` löscht das Komponentenelement mit Index `pos` aus der Liste und gibt `pos` zurück (dies ist dann genau die Indexposition des nachfolgenden Elementes). Falls die Methode für eine leere Liste aufgerufen wird oder der angegebene Index ungültig ist, gibt sie -1 zurück.

Die Methode `push_back(Komponente* k)` fügt das über den Zeiger `k` übergebene Komponentenobjekt, verpackt in der Struktur `KomponentenElement` (welche mit `new` anzulegen ist), am Ende der Liste ein.

Die Methode `begin()` liefert den Wert des Attributes `first` zurück.

Die Methode `end()` liefert immer den Wert `nullptr` zurück.

Im Destruktor muss die gesamte verkettete Liste mittels `delete` Anweisungen gelöscht werden.

II) Test der Implementierung:

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

   Man erstelle ein Hauptprogramm `P01Main.cpp`, in dem die Objekte vom Typ `Bohrung` (`b1, b2, b3, b4`) und die Objekte vom Typ `Fraesung` (`f1, f2, f3`) erstellt werden.

   Weiterhin füge man diese Objekte in der Reihenfolge `b1, f1, b3, f2, b4, f3, b2` in eine anfangs leere Liste vom Typ `KomponentenList` ein.

   Die Ausgabe dieser Liste sollte dann liefern:

```
Ausgabe der Liste
Bohrung: (1, 2), Durchmesser: 8
Fraesung mit Start: (2, 3) und Endpunkt: (4, 5), Durchmesser: 5
Bohrung: (10, 8), Durchmesser: 7
Fraesung mit Start: (6, 7) und Endpunkt: (9, 7), Durchmesser: 6
Bohrung: (7, 2), Durchmesser: 5
Fraesung mit Start: (17, 6) und Endpunkt: (13, 2), Durchmesser: 4
Bohrung: (2, 6), Durchmesser: 4
```

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

   In Moodle finden Sie für dieses erste Praktikum ein Qt-Creator-Projekt `P01Test`, indem Tests für Ihre implementierten Klassen mittels Google-Test-Framework vorbereitet sind.

   Fügen Sie Ihre Klassendateien diesem Projekt hinzu und führen Sie dann diese Projekt aus. Bestehen Ihre Klassen alle Tests? Können Sie nachvollziehen was getestet wird?

   Zur Installation des Google-Test-Frameworks finden Sie im Internet unter den Stichworten *Google Test Framework* und *Qt-Creator* ausführliche Anleitungen.

III) Reverse-Engineering mit dem CASE-Tool Visual Paradigm:

Starten Sie Visual Paradigm. Erstellen Sie sich zunächst ein neues Projekt (Menue-Punkt Projekt, Auswahl neu) und vergeben einen geeigneten Projektnamen.
Drücken Sie auf den Reiter `Tools`. Öffnen Sie das Menue `Code` und führen Sie dort `Reverse C++ Code` aus. Tragen Sie unter `Source Path` und unter `Cpp Path` das Verzeichnis ein, in dem sich Ihr Quellcode befindet und drücken OK.
Öffnen Sie im Menue am linken Rand den Reiter `Class Repository`. Bevor Sie mit dem Reverse-Engineering beginnen, stellen Sie durch Rechts-Click auf den Projektnamen unter Configure Programming Language sicher, dass C++ und nicht UML ausgewählt ist (ansonsten werden die Destruktor-Namen ohne führende Tilde angezeigt).

Durch Rechts-Click auf eine der Header-Dateien und die Auswahl
`Reverse HeaderDateiName.h to New bzw. Active Diagramm`
können Sie das UML-Klassendiagramm für die in der Header-Datei deklarierte Klasse/Struktur erzeugen und zum aktuellen Diagramm hinzufügen.
Kontrollieren Sie, ob das so für Ihren Code erstellte Diagramm mit dem in dieser Anleitung vorgegebenen Klassendiagramm übereinstimmt.
Leider unsterstützt Visual Paradigm noch nicht C++11. Sollten Sie schon die neuen Member-Initialisierungs-Listen, z.B.: `x{ xPos }`, verwendet haben, müssen Sie diese in die alte Form, z. B.: `x( xPos )`, umschreiben. Entsprechend sind alle anderen C++11 Neuerungen umzuschreiben, um Parse-Errors zu vermeiden.

Sie können sich Visual Paradigm for UML (Standard Ed.) wie folgt auf Ihrem eigenen Rechner installieren:

- Das Produkt kann über den Link
  `https://ap.visual-paradigm.com/hochschule-fur-angewandte-wissenschaften-munchen`
  heruntergeladen werden.
- Installieren Sie mithilfe des heruntergeladenen Assistenten das Produkt.
- Nach Abschluss der Installation, starten Sie die Anwendung.
- Wenn Sie die Anwendung das erste Mal starten, müssen Sie den **Lizenztyp Academic License** zuerst auswählen.
- Dann nutzen Sie zur Aktivierung den Aktivierungscode, der auf
  `https://ap.visual-paradigm.com/hochschule-fur-angewandte-wissenschaften-munchen`
  angegeben wurde und geben Sie Ihren Namen und Ihre hm.edu-Mailadresse an.
- Weitere Informationen finden Sie auf der Web-Seite:
  `https://www.ee.hm.edu/studienservices/softwarebezug/visual_paradigm_academic_program.de.html`