NACHNAME:	`Schlecht`	**SEMESTER:**	☐ **M5** ☐ **M6** ☐ M3 ☐ M4 ☐ M7
VORNAME:	`Net Wirklich`	**VERTIEFUNG:**	☐ **FV** ☐ **IM**

HILFSMITTEL:

- Ausdruck des vorab bekannt gemachten Quelltextes (16 Seiten = 8 Blätter) mit eigenen, handschriftlichen Ergänzungen

UNBEDINGT BEACHTEN:

- **Bevor** Sie mit der Bearbeitung beginnen, müssen die Angaben zur Person auf dieser Seite **vollständig** ausgefüllt sein.

GENERELLE VORGABEN:

- Der Quelltext der Klassen muss vollständig angegeben werden. Insbesondere sollen auch das `package` und benötigte `import`s enthalten sein.
- Es sind keinerlei Kommentare verlangt, weder `javadoc`-Kommentare noch andere.
- Programmier-Richtlinien (insbesondere Checkstyle) sind zu beachten. Bei der Testklasse dürfen aber *magic numbers* verwendet werden.

Aufgabe 1: (18 Punkte)

In dieser Aufgabe geht es um allgemeine Zusammenhänge.

a) Welche der folgenden Aussagen sind richtig?

	ja	nein
Beim Überschreiben muss die Parameterklammer gleich bleiben.	✓	
Beim Überschreiben muss der Rückgabetyp gleich bleiben.		✓
Jede Klasse besitzt eine öffentliche `clone` Methode.		✓
Jede Klasse besitzt eine öffentliche `getClass` Methode.	✓	
`private` Methoden können nicht überschrieben werden.	✓	
`private` Methoden können nicht überladen werden.		✓

b) Welche der folgenden Aussagen sind richtig?

	ja	nein
Die `toString` Methode kann eine ungeprüfte `Exception` werfen.	✓	
Die `toString` Methode kann eine geprüfte `Exception` werfen.		✓
Das Werfen einer ungeprüften `Exception` muss deklariert werden.		✓
Das Werfen einer geprüften `Exception` muss deklariert werden.	✓	
`CloneNotSupportedException` ist eine geprüfte `Exception`.	✓	
`IllegalArgumentException` ist eine geprüfte `Exception`.		✓

c) Welche der folgenden Empfehlungen sollten möglichst befolgt werden?

	ja	nein
Als Rückgabetyp ist eine `List` einem Array vorzuziehen.	✓	
Ein `Set` sollte möglichst parametrisiert (generisch) verwendet werden.	✓	
Statt einem leeren Array sollten Methoden besser `null` zurückgeben.		✓
Niemals sollte die Annotation `@SuppressWarnings` benutzt werden.		✓
`EnumSet` sollte nur benutzt werden, wo Performance zweitrangig ist.		✓

d) Welche der folgenden Empfehlungen sollten möglichst befolgt werden?

	ja	nein
Lokale Variablen sollten am Anfang der Methode deklariert werden.		✓
Der Gültigkeitsbereich lokaler Variablen sollte möglichst klein sein.	✓	
Nichtssagende Variablennamen wie `i` sollten auch in Schleifen vermieden werden.		✓
Nichtssagende Methodennamen sind konsequent zu vermeiden.	✓	
In Methodennamen sollten nur Kleinbuchstaben verwendet werden.		✓

Aufgabe 2: (34 Punkte)

Schreiben Sie eine konkrete Dekorierer-Klasse `Teil` für die Teilkurve einer `IKurve`. (Bekanntlich besteht eine Teilkurve aus denjenigen Punkten der Ausgangskurve, welche zu Parameterwerten in einem Teilintervall [a,b] $\subset$ [0,1] gehören.)

- Das Erzeugen einer Teilkurve soll nur über eine Fabrikmethode mit der Signatur `teilkurveVon(IKurve k, Intervall i)` möglich sein.
- Die Methoden zum Abbilden von Intervallen brauchen nicht korrekt implementiert zu werden. Sie sollen stattdessen eine `Exception` werfen, die anzeigt, dass die Methode nicht unterstützt wird.
- Die anderen Methoden der Schnittstelle sollen vollständig und korrekt implementiert werden.
- `System.out.println` soll eine sinnvolle Ausgabe liefern.

```
package pruefung2;


import pflichtuebung1.korrekt.Intervall;
import studienarbeit.*;


public final class Teil implements IKurve {
      private final IKurve mKurve;
      private final Intervall mIntervall;

      private Teil(IKurve k, Intervall i) {
            mKurve = k;
            mIntervall = i;
      }
      public static IKurve teilkurveVon(IKurve k,
                                        Intervall i) {
            IKurve tk = new Teil(k, i);
            return tk;
      }

```

```
        @Override
        public Intervall x(Intervall urbild) {
             throw new UnsupportedOperationException();
        }
        @Override
        public Intervall y(Intervall urbild) {
             throw new UnsupportedOperationException();
        }

        @Override
        public double x(double t) {
             double tg = globalerParameterZu(t);
             return mKurve.x(tg);
        }
        @Override
        public double y(double t) {
             double tg = globalerParameterZu(t);
             return mKurve.y(tg);
        }
        private double globalerParameterZu(double lokal) {
             double a = mIntervall.untereGrenze();
             double d = mIntervall.laenge();
             return a + lokal * d;
        }
```

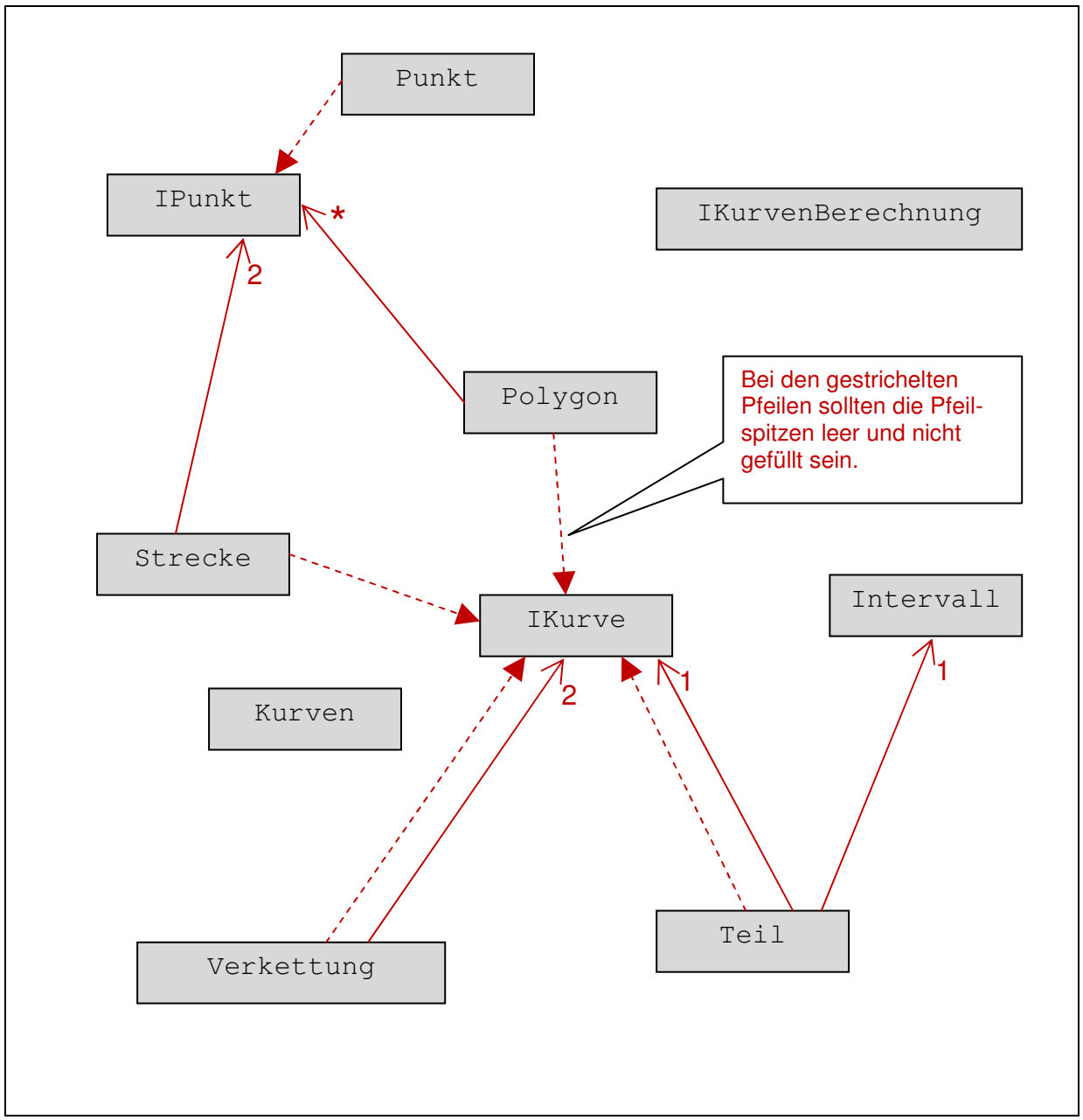
```
        @Override
        public String toString() {
             return "Teilkurve[" + mKurve + ","
                                 + mIntervall + "]";
        }

    }
```

Aufgabe 3: (18 Punkte)

Tragen Sie in das nachstehende Diagramm die Beziehungen zwischen den **angegebenen** Schnittstellen und Klassen in **UML**-Notation ein.

- Die Beziehungen sollen gerichtet sein. Wo möglich und sinnvoll soll die Kardinalität angegeben werden.
- Unterscheiden Sie zwischen der Implementierung von Schnittstellen und echter Vererbung.
- Unterscheiden Sie **nicht** zwischen Kompositionen, Aggregationen und einfachen Assoziationen.
- Schwache Abhängigkeiten, bei denen keine konkreten Objekte betroffen sind, dürfen **nicht** eingezeichnet werden.



Aufgabe 4: (20 Punkte)

Schreiben Sie eine rudimentäre (**JUnit 4**) Testklasse für die Klasse `Teil` mit einer einzigen Methode.

- Geprüft werden soll die zu [½,1] gehörende Teilkurve der Strecke von (-100,200) nach (100,0).
- Konkret überprüft werden soll der zum Parameterwert ½ gehörende Kurvenpunkt der Teilkurve, sowie die Bilder `x([0,1])` und `y([0,1])`.
- Wo möglich, soll auf exakte Übereinstimmung getestet werden.

```
package pruefung2.hilfsmittel;

import org.junit.Test;

import static org.junit.Assert.*;

import pflichtuebung1.korrekt.Intervall;
import studienarbeit.*;
import pruefung2.*;

import static studienarbeit.IKurve.INTERVALL;
import static pruefung2.Kurven.kurvenPunkt;
import static pruefung2.Teil.teilkurveVon;


public class TestTeil {

	@Test
	public void testeStreckeEinfach() {

```

```
            IPunkt p1 = new Punkt(-100., 200.);
            IPunkt p2 = new Punkt(100., 0.);
            Strecke strecke = new Strecke(p1, p2);
            Intervall oben = INTERVALL.obereHaelfte();
            IKurve teil = teilkurveVon(strecke, oben);

            IPunkt pSoll = kurvenPunkt(strecke, 0.75);
            assertEquals(pSoll, kurvenPunkt(teil, 0.5));

            Intervall iSoll = new Intervall(0., 100.);
            assertEquals(iSoll, teil.x(INTERVALL));
            assertEquals(iSoll, teil.y(INTERVALL));

        }

    }
```