Fragen, Antworten, Kommentare zur aktuellen Vorlesung

Frage: Ich glaube, dass ich sehr viel Zeit für die Praktikumsaufgaben benötige.

Das ist generell ok, wobei die Bearbeitungszeit meist unmittelbar von den eigenen Erfahrungen abhängt. Da die Fehlersuche unterschiedlich lange dauern kann, ist gerade bei der Programmierung die Bearbeitungszeit schwer zu planen. Generell wissen Sie, dass ein Drittel des Semesters sich mit Programmierung beschäftigt. Da Sie ein Vollzeitstudium machen, entspricht dies mindestens 13 Arbeitsstunden pro Woche. Dies umfasst alle Arbeitszeiten von Videos besorgen, diese anschauen und parallel mit zu programmieren, Fragen mit Anderen zu diskutieren, im Praktikum zu sein und natürlich die Aufgaben zu bearbeiten. Falls das nach viel klingt, lesen Sie sich bitte die formale Beschreibung von Leistungspunkten (ein Punkt entspricht 30 Arbeitsstunden einer befähigten durchschnittlichen studierenden Person) durch und berechnen Sie die Arbeitszeit mit einer realistischen Verteilung von vorlesungs- und vorlesungsfreier Zeit.

Hinweis: Ich habe beim Praktikum vereinzelt festgestellt, dass der Umgang mit Vokabeln, wie Klasse und Objektmethode nicht ganz sauber ist. Denken Sie daran die Begriffe präzise zu nutzen, da so andere Personen genau wissen, was sie meinen. Das ist bei „Dings in Klammern“ etwas schwierig. Für Begriffe gibt es ein Dokument auf der Veranstaltungsseite: http://kleuker.iui.hs-osnabrueck.de/querschnittlich/CodingGuidelinesUndGlossar.pdf .

Zum Lernen der Begriffe können Sie auf Ihnen bekannte Lernmethoden zurückgreifen. Ein interessanter Ansatz ist es, sich die Begriffe laut selbst zu erklären. Zeichen Sie dies einfach in Zoom mal auf und schauen Sie sich das Ergebnis zur Selbstkontrolle an.

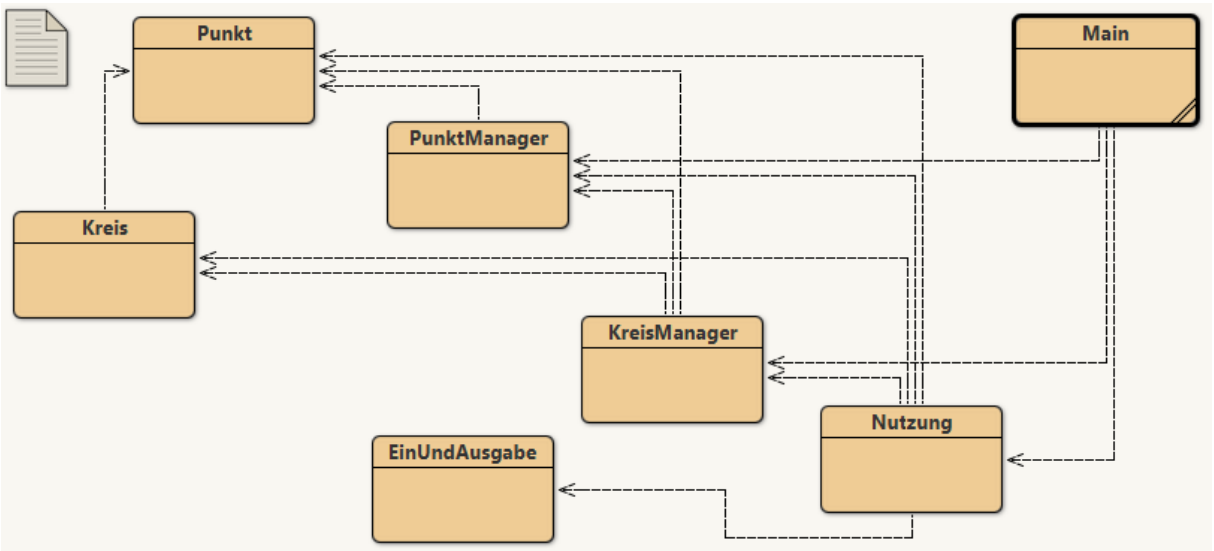
Info: Unsere Klausur ist (leider erst) für den 31.1.23 geplant, weitere Details und eine Musterklausur kommen später. Wieso „leider erst“? Der späte Termin suggeriert, dass durch fleißiges Lernen nach der Vorlesungszeit der Stoff nachgeholt werden kann. Meine Erfahrungen zeigen, dass das nie klappt, ich keine Person kenne, die während der Praktika enorme Schwierigkeiten hatte und dann die Klausur mitgeschrieben und bestanden hat. Falls es so jemanden geben sollte, wäre ich danach an einem Erfahrungsaustausch sehr interessiert.

Früher waren Klausuren einfach in den zwei Wochen nach der Vorlesungszeit. So war klar, dass ein Nachlernen kaum möglich ist. Da in der Programmierklausur auf Papier programmiert wird, ist auch klar, dass ein Auswendiglernen sinnlos ist. „Auf Papier programmieren“ klingt zunächst merkwürdig, hat aber einen didaktischen Vorteil. Vor der Lösung muss ein Verfahren im Kopf entstehen, mit dem die Aufgabe gelöst wird. Es kann nicht mit Try-And-Error versucht werden zum Ergebnis zu kommen. Try-and-Error ist genau der Ansatz den gute Leute in der Programmierung nicht nutzen, da so oft versteckte Fehler zurückbleiben. Es wird schrittweise ein komplexeres System entwickelt und zwischenzeitlich getestet. Diese Tests allerdings fallen auf Papier weg. Dafür spielt beim Schreiben die exakte Syntax (vergessene Klammer, vergessenes Semikolon) keine Rolle und ist die Idee zielführend, die Umsetzung aber nicht, gibt es abhängig vom Fehler 50-80% der Punkte.

Ausflug: Bei der Praktikumsaufgabe zur Erstellung der Klassen Punkt und Kreis stellt sich die Frage, wer erstellt eigentlich wann Objekte. Für Personen am absoluten Anfang der Programmiererfahrungen ist die nachfolgende Überlegung ein Ausflug in deutlich später folgende Themen, sie sollten aber mindestens den entstehenden Code verstehen. Der Startpunkt der Diskussion ist der folgende Konstruktor.

```
class Kreis {
  Kreis(Punkt aufhaengepunkt, int radius){ // ueblicher Konstruktor
    this.aufhaengepunkt = aufhaengepunkt;
    this.radius = radius;
  }
```

Neben den Experimenten, die in der Veranstaltung im Mittelpunkt stehen und in denen an beliebigen Stellen Objekte erstellt werden, stellt sich die Frage, wie es später in der Praxis aussieht. Das zentrale Ziel in der Praxis ist immer wartbarer und erweiterbarer Code. Ein Teilziel davon ist die Möglichkeit kleine funktionale Veränderungen an Programmen vornehmen zu können ohne viele Klassen bearbeiten zu müssen. Findet dann eine Objekterstellung einer Klasse, also Aufruf von new, an vielen Stellen statt und soll das Verhalten bei der Objekterstellung angepasst werden, kann dies schnell sehr aufwändig werden. Ein fiktives Beispiel ist, dass bei der Erstellung eines Punktes geklärt werden soll, ob der Aufrufer überhaupt die Rechte dazu hat. Konkret werden diese Rechte wieder mit einem Objekt verwaltet, dass bei Erstellung genutzt werden soll. Eine nullte Idee könnte sein, diese Überprüfung in den Konstruktor einzubauen. Da aber Klassen nur eine Kernaufgabe haben sollen, ist das ein falscher Ansatz. Ein besserer Ansatz ist es, die Erstellung von Objekten jeweils an einer Stelle, genauer in er Klasse zu fokussieren. Hierzu werden gerne Verwaltungs- oder engl. Manager-Klassen genutzt.



Dies soll anhand eines Beispiels erklärt werden. Das zugehörige BlueJ-Diagramm wie folgt aus, nur die Klassen Punkt und Kreis werden als bekannt vorausgesetzt.

Die Erzeugung von Punkt-Objekten übernimmt der PunktManager. Jedes Objekt, das einen Punkt haben will, muss diesen Manager nutzen. Damit gibt es später nur eine Stelle, an der Punkt-Objekte erzeugt werden.

```
class PunktManager {

  Punkt erzeugePunkt(int x, int y){
    return new Punkt(x, y);
```

```
  }
}
```

Ähnlich arbeitet der KreisManager, der allgemein Kreise verwalten und damit auch ihre Erzeugung übernehmen wird. Hier ist es diskutabel, ob zur Objekterzeugung bereits ein Punkt existieren muss oder dieser hier erzeugt werden kann. Die zweite Variante kann „zur Bequemlichkeit“ auch angeboten werden. Es ist am Code erkennbar, dass ein Manager einen anderen nutzen kann.

```
class KreisManager {
  PunktManager punktmanager;

  KreisManager(PunktManager pm){
    this.punktmanager = pm;
  }

  Kreis erzeugeKreis(int x, int y, int radius){ // evtl. aus Bequemlichkeit
    Punkt tmp = this.punktmanager.erzeugePunkt(x, y);
    return new Kreis(tmp, radius);
  }

  Kreis erzeugeKreis(Punkt punkt, int radius){ // der wichtige
    return new Kreis(punkt, radius);
  }
}
```

Konsequent stellt sich die Frage, wer die Manager erzeugt. Im einfachsten und oft vorkommenden Fall gibt es eine zentrale Klasse mit einer zentralen Methode, in der die Manager-Objekt erzeugt und an alle nutzenden Klassen, auch zur späteren Weiterverteilung, übergeben werden. Dies zeigt die folgende Beispielklasse.

```
class Main {
  void main(){
    PunktManager punktmanager = new PunktManager();
    KreisManager kreismanager = new KreisManager(punktmanager);
    Nutzung nutzung = new Nutzung(punktmanager, kreismanager);
    nutzung.nutzen();
  }
}
```

Die nutzenden Klassen bekommen die Manager übergeben, werden sie niemals selbst erzeugen und können die Manager über die angebotenen Methoden nutzen. Dieser Ansatz wird auch als „Dependency Injection“ bezeichnet, benötigte Objekte werden über Konstruktoren oder set-Methoden übergeben. Eine Beispielnutzung kann wie folgt aussehen, die erwähnte Klasse ToStringer von der Veranstaltungsseite wird hier als eventuell irritierende Spielerei mal gezeigt und spielt bei den eigentlichen Überlegungen keine Rolle.

```
import gsdet.toStringer.ToStringActivator; // loeschen, wenn nicht in kleukersSEU

class Nutzung {
  PunktManager punktmanager;
  KreisManager kreismanager;

  Nutzung(PunktManager pm, KreisManager km){
    this.punktmanager = pm;
    this.kreismanager = km;
  }

  void nutzen(){
    Punkt pu = this.punktmanager.erzeugePunkt(26, 2);
    Kreis kr = this.kreismanager.erzeugeKreis(pu, 20);
```

```
    EinUndAusgabe io = new EinUndAusgabe();
    io.ausgeben(pu.getX() + " " + pu.getY() + "\n");
    io.ausgeben(kr.getAufhaengepunkt().getX()
      + " " + kr.getAufhaengepunkt().getY()
      + " " + kr.getRadius() + "\n");
  }
}
```

Die Ausgabe des Programms sieht wie folgt aus.

```
26 2
26 2 20
```