Funktionale und objektorientierte Programmierkonzepte
Übungsblatt 05

TECHNISCHE UNIVERSITÄT DARMSTADT

Prof. Karsten Weihe

Wintersemester 23/24	v1.2.2
Themen:	Vererbung, Objektorientierung
Relevante Foliensätze:	03a und 03b
Abgabe der Hausübung:	01.12.2023 bis 23:50 Uhr

Hausübung 05 — Gesamt: 32 Punkte

EDV-Zoo

Beachten Sie die Seite *Verbindliche Anforderungen für alle Abgaben* im Moodle-Kurs.

Verstöße gegen verbindliche Anforderungen führen zu Punktabzügen und können die korrekte Bewertung Ihrer Abgabe beeinflussen. Sofern vorhanden, müssen die in der Vorlage mit `TODO` markierten `crash`-Aufrufe entfernt werden. Andernfalls wird die jeweilige Aufgabe nicht bewertet.

Die für diese Hausübung relevanten Verzeichnisse sind `src/main/java/h05` und ggf. `src/test/java/h05`.

Verbindliche Anforderung: Dokumentieren Ihres Quelltexts

Alle von Ihnen deklarierten Klassen, Interfaces, Enumerationen und Methoden (inklusive Konstruktoren), die nicht `private` sind, *müssen* für diese Hausübung mittels JavaDoc in Englisch oder alternativ Deutsch dokumentiert werden. Für jede korrekte Deklaration ohne Dokumentation verlieren Sie jeweils einen Punkt.

Beachten Sie die Seite *Hausübungen → Dokumentieren von Quelltext* im Studierenden-Guide.

Verwenden Sie in Ihrem Quelltext 1:1 die auf diesem Übungsblatt gewählten Identifier! Andernfalls wird die jeweilige Aufgabe nicht automatisiert bewertet.

Einleitung

Im Rahmen der fünften Hausübung werden Sie sich damit befassen, ein Computersystem abzubilden und dieses in Hinblick auf bestimmte Merkmale untersuchen.

Im Rahmen dieser Übung lösen Sie sich von *FopBot* und erstellen zudem alle Dateien des Quellcodes selbst. Falls nicht anders angegeben, lösen Sie die Aufgaben in Package `h05`

> Verwenden Sie in Ihrem Quelltext 1:1 die auf diesem Übungsblatt gewählten Identifier! Andernfalls wird die jeweilige Aufgabe nicht automatisiert bewertet.

> Wenn die Rede davon ist, dass Klassen, Interfaces oder Enumerationen *erstellt* werden sollen, müssen zuerst die dazugehörigen Dateien in `src/main/java/h05` (sofern nicht anders angegeben) erstellt werden.

H1: Komponenten Modellierung — 13 Punkte

Alle folgenden Teilaufgaben müssen in Package `h05.model` implementiert werden.

H1.1: Interface `Component` — 1 Punkt

Schreiben Sie in eine Datei `Component.java` ein `public`-Interface `Component`. Das Interface besitzt die parameterlose Methode `getPrice`, welche den Rückgabetyp `double` hat. Sie werden dieses Interface nutzen, um die verschiedenen Komponenten eines Computersystems zu modellieren.

H1.2: Enumerationen — 2 Punkte

Schreiben Sie in einer Datei `Socket.java` eine `public`-Enumeration mit den Konstanten `AM4` und `LGA1700`.

Schreiben Sie in einer Datei `PeripheralType.java` eine `public`-Enumeration mit den Konstanten `GPU` für eine Grafikkarte, `ETHERNET` für die Netzwerkanbindung und `TPU` für *Tensor*-Prozessoren. Tensor-Prozessoren sind anwendungsspezifische Chips um maschinelles Lernen zu beschleunigen.

H1.3: `CPU` modellieren — 4 Punkte

Erstellen Sie in einer Datei `CPU.java` eine `public`-Interface `CPU`, welche das Interface `Component` erweitert. Diese Klasse soll genutzt werden um einen Prozessor zu modellieren, der einen bestimmten Socket, eine bestimmte Anzahl an Kernen und eine Taktfrequenz, sowie einen Preis besitzt.

Erstellen Sie in `CPU` die parameterlose Methoden

- `getSocket` mit Rückgabetyp `Socket`,
- `getCores` mit Rückgabetyp `int`,
- und `getFrequency` mit Rückgabetyp `double`.

Erstellen Sie außerdem eine `public`-Klasse `CPUImpl` in einer Datei `CPUImpl.java`, die das zuvor erstellte Interface `CPU` implementiert. Die Klasse besitzt eine `private`-Objektkonstanten `socket` vom Typ `Socket`, `numOfCores` von Typ `int`, `frequency` von Typ `double`, `price` von Typ `double`. Die jeweilige `get`-Methode liefert den jeweiligen Wert bzw. Objekt zurück.

Außerdem besitzt `CPUImpl` einen `public`-Konstruktor, der formale Parameter `socket` von Typ `Socket`, `numOfCores` von Typ `int`, `frequency` von Typ `double` und `price` von Typ `double` in genau dieser Reihenfolge besitzt. Die Parameter sollen genutzt werden, um die oben beschriebenen Konstanten zu initialisieren.

H1.4: `Memory` modellieren — 1 Punkt

Analog zu Aufgabe H1.3 modellieren Sie in dieser Aufgabe den Hauptspeicher eines Computers. Hierzu erstellen Sie das `public`-Interface `Memory` in einer Datei `Memory.java`, welche `Component` erweitert. Diese besitzt analog zu `CPU` eine Methode `getCapacity` von Typ `char`. Der Wert dieser Methode wird hierbei als die Kapazität des Speichers in Gigabyte interpretiert.

Erstellen Sie außerdem eine dazugehörige Implementierung in Form einer `public`-Klasse `MemoryImpl` in einer Datei `MemoryImpl.java`, die `Memory` implementiert. Die Klasse `MemoryImpl` implementiert alle vordefinierten Methoden. Dazu besitzt sie die `private`-Objektkonstante `capacity` vom Typ `char`, wessen aktueller Wert in einer `get`-Methode zurückgeliefert wird. Außerdem legen Sie, wie in der vorherigen Teilaufgabe auch, eine `private`-Objektkonstanten `price` vom Typ `double` an, um den Preis des Speichers abzuspeichern, und die jeweilige `get`-Methode, um den Wert auslesen zu können. Der `public`-Konstruktor besitzt die gleichnamigen Parameter `capacity` und `price` vom jeweiligen Typ, sodass die Konstanten `capacity` und `price` initialisiert wird.

H1.5: `Peripheral` modellieren — 1 Punkt

Zuletzt wird analog auch noch Peripherie des Computers durch ein `public`-Interface `Peripheral` in der Datei `Peripheral.java` modelliert, welches von `Component` erbt.

Sie gehen hierbei analog zu `CPU` und `Memory` vor. Die zu modellierende Eigenschaft von `Peripheral` ist der jeweilige `PeripheralType`. Definieren Sie daher die parameterlose Methode `getPeripheralType`, die ein `PeripheralType` zurückliefert.

Implementieren Sie `Peripheral` in einer `public`-Klasse `PeripheralImpl` in einer Datei `PeripheralImpl.java`. Die Klasse implementiert hierbei das Interface `Peripheral`. `PeripheralImpl` besitzt eine `private`-Objektkonstante `peripheralType` von Typ `PeripheralType` und `price` von Typ `double`. Der `public`-Konstruktor besitzt einen Parameter `peripheralType` von Typ `PeripheralType` und einen `price` von Typ `double`. Der Konstruktor setzt hierbei wie gewohnt die Konstanten `peripheralType` und `price`. Die jeweilige `get`-Methode liefert den jeweiligen Wert zurück.

H1.6: `Component` implementieren — 4 Punkte

In dieser Aufgabe werden Sie zurück auf die vorherigen Teilaufgaben blicken und sehen wie man die Implementierung dieser verbessern kann. Wie Ihnen vielleicht aufgefallen ist, weist `CPUImpl`, `MemoryImpl` und `PeripheralImpl` eine Redundanz bezüglich der Methode `getPrice` des Interfaces `Component` auf. Alle drei Klassen definieren einen festen Preis für die Komponente und die `getPrice` liefert jeweils diesen Preis zurück. In dieser Aufgabe werden Sie dies in Form der Klasse `PurchasedComponent` heraus faktorisieren. Diese Klasse repräsentiert eine Komponente, die einen festen, bereits bei der Erstellung definierten, Preis besitzt.

Erstellen Sie in einer Datei `PurchasedComponent.java` eine `public`-Klasse `PurchasedComponent`, welche

- das Interface `Component` implementiert,
- eine `private`-Objektkonstante `price` vom Typ `double` besitzt,
- einen `public`-Konstruktor mit genau einem `double`-Parameter hat und die Konstante `price` mit diesem Wert initialisiert
- und die `getPrice`-Methode von `Component` so implementiert, dass der Wert von `price` zurückgeliefert wird.

Des Weiteren soll es nicht möglich sein ein Objekt von `PurchasedComponent` zu erzeugen.

Passen Sie anschließend `CPUImpl`, `MemoryImpl` und `PeripheralImpl` so an, dass diese von `PurchasedComponent` abgeleitet werden. Entfernen Sie für diese Klassen die Objektkonstante `price` und die konkrete Implementierung von `getPrice`. Außerdem müssen Sie den Konstruktor so anpassen, dass der Konstruktor der Basisklasse mit den richtigen Werten aufgerufen wird.

> **Hinweis:**
>
> Es passiert oft, dass man in frühen Stadien bei der Entwicklung von Software (gilt auch für andere Bereiche) Entscheidungen trifft, die sich später als nicht-optimal herausstellen. Oft kristallisieren sich Verbesserungen erst während der Implementierung raus. Der Prozess bei dem man die interne Struktur des existierenden Programmcodes verbessert ohne dabei das Verhalten des Codes zu ändern wird auch *Refactoring* bezeichnet.

H2: System Modellierung — 5 Punkte

Um ein minimal funktionierendes Computersystem zu modellieren fehlt zuletzt noch das Mainboard, welches die unterschiedlichen Komponenten, also Prozessor, Hauptspeicher und Peripherie, verbindet.

Definieren Sie hierzu zunächst ein `public`-Interface `Mainboard` in einer Datei `Mainboard.java` in Package `h05.model`, welches selbst eine Komponente ist und daher von Interface `Component` erbt.

Definieren und implementieren Sie anschließend eine `public`-Klasse `MainboardImpl` in der Datei `MainboardImpl.java` im selben Package, welche `Mainboard` implementiert. Leiten Sie `MainboardImpl` außerdem von `PurchasedComponent` ab, damit Sie `getPrice` wiederverwenden.

Es wird angenommen, dass ein Mainboard Platz für genau einen Prozessor, eine fest definierte Anzahl an Speicher-Slots und eine feste Anzahl an Peripherie-Slots hat. Die Attribute der Klasse ergeben sich hierdurch zu `cpu` von Typ `CPU`, `memories` von Typ `Memory[]` und `peripherals` von Typ `Peripheral[]`.

Der `public`-Konstruktor besitzt die Parameter `socket` von Typ `Socket`, `numberOfMemorySlots` von Typ `int`, `numberOfPeripheralSlots` von Typ `int` und `price` von Typ `double`. Der `price` wird genutzt um den Konstruktor von `PurchasedComponent` aufzurufen. Speichern Sie den Wert von `socket` in einem geeigneten Attribut ab, da dieses später benötigt wird, um zu prüfen, ob das Mainboard für einen bestimmten Prozessor geeignet ist Nutzen Sie `numberOfMemorySlots` bzw. `numberOfPeripheralSlots` um `memories` bzw. `peripherals` mit jeweiliger Länge zu initialisieren.

Erstellen Sie eine `public`-Methode `addCPU`, die einen formalen Parameter von Typ `CPU` nimmt und `boolean` zurückliefert. Der übergebene Prozessor wird genau dann in das Mainboard gesetzt, wenn er nicht `null` ist, bisher das Attribut `cpu` `null` ist und der Sockel des Prozessors mit dem des Mainboards übereinstimmt. Sind diese Bedingungen erfüllt, wird der Prozessor im Attribut `cpu` gespeichert. Wurde der Prozessor erfolgreich zum Modell hinzugefügt, so liefert die Methode `true` zurück, andernfalls `false`.

Die Methode `addMemory` bzw. `addPeripheral` bekommt einen formalen Parameter von Typ `Memory` bzw. von Typ `Peripheral` und liefert `boolean` zurück. Das übergebene `Memory`- bzw. `Peripheral`-Objekt soll hierbei an die nächste freie Stelle in `memories` bzw. `peripherals` eingefügt werden. Ist dies erfolgreich, soll `true` zurückgegeben werden, andernfalls, also genau dann, wenn es keinen freien Slot mehr gibt, `false`.

H3: Bewertung

Durch die vorgehenden Aufgabe ist es möglich ein Modell eines Computersystem zusammenzustellen, jetzt fehlt noch die Funktionalität, dass ein System anhand von verschiedenen Kriterien bewertet werden kann. Das Ziel ist, dass verschiedene Systeme durch das Bewerten, also zum Beispiel dem Zuordnen eines *Scores*, verglichen werden können.

H3.1: Interface `ComponentRater` — 1 Punkt

Erstellen sie ein `public`-Interface `ComponentRater` in einer Datei `ComponentRater.java`. Das Interface besitzt folgende rückgabelose Methoden

- `consumeMainboard` mit Parameter `mainboard` von Typ `Mainboard`,
- `consumeCPU` mit Parameter `cpu` von Typ `CPU`,
- `consumeMemory` mit Parameter `memory` von Typ `Memory`,
- und `consumePeripheral` mit Parameter `peripheral` von Typ `Peripheral`.

> **Hinweis:**
>
> Das Programm wurde hierbei so aufgebaut, um das sogenannte *Besucher-* bzw. *Visitor*-Entwurfsmuster zu verwenden.
>
> In der Softwareentwicklung ist es typisch verschiedene Entwurfsmuster zu nutzen. Entwurfsmuster sind bewährte Lösungsansätze für wiederkehrende Probleme in der Softwareentwicklung. Sie lassen sich als Schablonen zum Lösen des Problems nutzen.
>
> Das Visitor-Muster dient hierbei zur Kapselung von Operationen. Die Operationen, die in dieser Hausübung gekapselt sind, sind die jeweiligen `consume`*-Methoden, die die einzelnen Komponenten bewerten. Durch verschiedene konkrete Implementierungen eines abstrakten Visitors lassen sich so verschiedene Verhalten erzeugen, wie Sie auch in den folgenden Teilaufgaben sehen werden, während die Implementierung in den einzelnen Komponenten unverändert bleibt.

H3.2: Interface `Configuration` — 3 Punkte

Schreiben Sie ein `public`-Interface `Configuration` in einer Datei `Configuration.java`. Diese soll eine rückgabelose Methode `rateBy` haben, die einen Parameter `rater` von Typ `ComponentRater` nimmt.

Passen Sie zunächst das Interface `Mainboard` so an, dass `Mainboard` das Interface `Configuration` erweitert, da wir ein Mainboard als eine vollwertige Konfiguration ansehen. Anschließend müssen Sie Ihre Implementierung für `MainboardImpl` so anpassen, dass sie die Methode `rateBy` von `Configuration` implementiert. Die Implementierung von `rateBy` sieht hierbei so aus, dass für jede Komponente die jeweilige `consume`-Methode, also Mainboard, Prozessor, alle Speicher und jede Peripherie, aufgerufen wird. Achten Sie hierbei darauf, dass die `consume`-Methode nur dann aufgerufen wird, wenn die Referenz wirklich auf ein Objekt zeigt, also ungleich `null` ist.

H3.3: Verschiedene Implementierungen — 3 Punkte

Im Folgenden sollen Sie nun zwei verschiedene Implementierungen für `ComponentRater` erstellen.

Zunächst implementieren Sie die `public`-Klasse `TotalCostRater` in einer Datei `TotalCostRater.java`, welche das Interface `ComponentRater` implementiert. Wie der Name erahnen lässt, soll dieser `ComponentRater` genutzt werden um die Gesamtkosten einer Konfiguration bestimmen zu können. `TotalCostRater` soll eine `public`-Methode `getTotalPrice` besitzen, die keinen Parameter nimmt und `double` zurückliefert. Hierbei soll diese Methode den Wert eines Attributs `cost` von Typ `double` zurückliefern. Dieses Attribut wird genutzt, um den kumulierten Wert der Konfiguration zu speichern. Implementieren sie die vier `consume`-Methoden so, dass der Wert von `cost` entsprechend aktualisiert wird.

Anschließend implementieren Sie nun eine `public`-Klasse `MachineLearningRater` in einer Datei `MachineLearningRater.java`, die analog auch wieder das Interface `ComponentRater` umsetzt. Der `MachineLearningRater` soll verwendet werden, um Systeme hinsichtlich der Fähigkeit zum Maschinellen Lernen zu bewerten. Zur Vereinfachung berechnen nutzen wir die kumulierte Kapazität des Hauptspeichers, um zu prüfen, wie schnell sich ein Lernalgorithmus auf einem System ausführen lassen kann. Nutzen Sie `int` zum Speichern der Kapazität, um Überläufe zu vermeiden. Überlegen Sie sich selbst welche Attribute Sie anfügen und wie Sie die `consume`-Methoden realisieren, sodass die nötigen Informationen gesammelt werden.

Die `public`-Methode `checkModel` besitzt einen Parameter `modelSize` von Typ `int`, der die Größe des Modells des Lernalgorithmus beschreibt, und liefert den Score als `double` zurück. Der Score errechnet sich durch:

$$\texttt{Score} = \frac{\texttt{Gesamtkapazität}}{\texttt{Modellgröße}} \cdot \textit{TPU Faktor}$$

$$\textit{TPU Faktor} = 100 \cdot (100 - 1) \cdot 1.02^{-\texttt{Anzahl TPUs}}$$

H4: Testen

In dieser Aufgabe werden Sie Ihre Implementierung testen. Dies sollen Sie in der `main`-Methode in der Klasse `Main` implementieren.

H4.1: Explizite Konfiguration testen — 2 Punkte

Erstellen Sie folgende Objekte der Konfiguration

- ein Mainboard mit `AM4`-Sockel, zwei Speicherslots und zwei Peripherieslots, es kostet 100 Euro
- einen Prozessor mit `AM4`-Sockel, zehn Kernen, einer Frequenz von `3.3` Gigahertz und einem Preis von 300 Euro
- einem Speicher mit acht Gigabyte und einem Preis von 60 Euro
- einer Grafikkarte mit einem Preis von 300 Euro

und fügen Sie die einzelnen Komponenten mit der jeweiligen `add`-Methode zum Mainboard hinzu.

Bewerten Sie diese Konfiguration jeweils mit `TotalCostRater` und `MachineLearningRater` und bestimmen Sie außerdem per Hand einen Referenzwert. Zum Vergleichen dieser Werte nutzen Sie die Methode `testWithinRange` der `AlgoUtils`.

Anmerkung:

Spielen Sie hierbei mit verschiedenen Werten der Konfiguration, um Ihre Implementierung zu testen. Denken Sie vor der Abgabe jedoch daran, die Werte wieder auf die geforderte Konfiguration zu ändern.

H4.2: `VirtualMemory` implementiert `Memory` — 2 Punkte

Erstellen Sie eine `public`-Klasse `VirtualMemory` in einer Datei `VirtualMemory.java` in Package `h05.model` und lassen Sie diese `Memory` implementieren. Ein `VirtualMemory` soll hierbei ein alternatives Konzept für einen Hauptspeicher darstellen. Man muss kein Geld für einen Speicherriegel mit fester Größe ausgeben, sondern kann die Kapazität dynamisch während der Laufzeit anpassen. Daher weist ein `VirtualMemory` auch kein festen Preis auf, sondern ein variablen Preis, der sich proportional zum angeforderten Speicher ändert.

Der `public`-Konstruktor besitzt nur einen Parameter `costPerGigaByte` von Typ `double`, der den Basispreis für je ein Gigabyte allokierten Speicher angibt. Implementieren Sie abgesehen von `getPrice` und `getCapacity` außerdem eine parameterlose `public`-Methode `setCapacity` mit einem Parameter von Typ `char`, die es erlaubt die Größe des Speichers dynamisch zu ändern.

Anmerkung:

Erstellen Sie eine Konfiguration, die ein `VirtualMemory` besitzt und bewerten Sie diese mittels der `ComponentRater`. Passen Sie die Größe des allokierten Speichers mittels eines geeigneten Aufrufs von `setCapacity` auf und bewerten Sie diese angepasste Konfiguration erneut. Überprüfen Sie, ob das Ergebnis mit Ihrer Überlegung übereinstimmt.

H4.3: Server — 3 Punkte

Erstellen Sie eine `public`-Klasse `ServerCenter` in einer Datei `ServerCenter.java`. Diese Klasse soll eine `public`-Methode `addMainboard` besitzen, die ein `Mainboard` als Parameter nimmt und nichts zurückliefert. Stellen Sie hierbei sicher, dass das `ServerCenter` mit mindestens 10000 `Mainboard`s auskommt. Dies ist ein durchaus realistischer Wert für große Rechenzentren. Implementieren Sie das Interface `Configuration` in `ServerCenter`. Die Methode `rateBy` implementieren Sie hierbei so, dass für jedes `Mainboard` die `rateBy`-Methode des jeweiligen `Mainboard`s mit dem übergebenen `rater` aufgerufen, also die Anfrage weitergeleitet bzw. *delegiert*, wird.

Anmerkung:

Erstellen Sie eine Konfiguration eines `ServerCenter`s und fügen Sie verschiedene `Mainboard`s hinzu. Bewerten Sie die Konfiguration mittels der beiden `ComponentRater` und prüfen Sie, ob die Ausgabe korrekt ist.