Objektorientierte Programmiertechniken
LVA 185.A01, VU, 3 Ects, 2013/2014 W

4. Übungsaufgabe

Themen:

Untertypbeziehungen, Zusicherungen

Termine:

Ausgabe: 30.10.2013
Abgabe: 13.11.2013, 12:00 Uhr

Abgabeverzeichnis:

Aufgabe4

Programmaufruf:

java Test

Grundlage:

Skriptum, Schwerpunkt auf Kapitel 2

Aufgabe

Welche Aufgabe zu lösen ist:

Kranichkirchener sind jetzt schon für ihre hohe Software-Kompetenz bekannt. Dennoch finden sich auch dort immer wieder Programme, die wegen trivialer Ursachen schwer lesbar sind. Eine bessere Unterstützung für die Formatierung von Kommentaren könnte Abhilfe schaffen.

Folgendes Interface ist vorgegeben:

```
public interface Prettifier {
   String pretty(String prog);
   /* Result corresponds to the Java program in prog,
    * but with more, less or prettier comments.
    * This method has no side-effects.
    */
}
```

Es sollen folgende Untertypen von *Prettifier* als Klassen, abstrakte Klassen oder Interfaces erstellt werden:

- Objekte des Typs *Copyrighter* stellen durch Ausführung von *pretty* einen Kommentar ganz an den Anfang des Programms, der Copyright-Informationen, die Namen der Autoren und das Datum enthält. Es gibt jeweils eigene Methoden um Autoren hinzuzufügen, Autoren zu entfernen und das Datum sowie Copyright-Informationen zu setzen. Weitere Methoden zur Änderung des Kommentars können auch vorhanden sein. Änderungen durch diese Methoden wirken sich auf den nächsten Aufruf von *pretty* aus.
- Objekte des Typs *Adder* fügen durch *pretty* Kommentare in das Programm ein. Über den Konstruktor lässt sich der im Kommentar enthaltene Text (kann auch mehrere Zeilen umfassen oder leer sein) sowie die Position dieses Kommentars festlegen – ganz am Anfang oder Ende des Programms, oder vor oder nach jeder Zeile, die eine bestimmte Zeichenfolge enthält. Dabei werden nur Zeichenfolgen berücksichtigt, die nicht in einem Kommentar oder String enthalten sind. So kann beispielsweise der Text “Anfang des Klassenrumpfes” nach jeder Zeile mit der Zeichenfolge “class” (außerhalb eines Kommentars oder Strings) verwendet werden um den Anfang jeden Klassenrumpfs durch eine Kommentarzeile

  ```
  /*Anfang des Klassenrumpfes*/
  ```

  zu markieren. Es gibt eine Methode um den Text im Kommentar zu ändern, der sich auf den nächsten Aufruf von *pretty* auswirkt. Aber es gibt keine Möglichkeit, nach Erzeugung eines Objekts die Position des einzufügenden Kommentars zu ändern.
- Objekte des Typs *Eraser* stellen das Gegenstück zu *Adder* dar: Je nach der über den Konstruktor bestimmten Position – ganz am Anfang oder Ende des Programms, oder vor oder nach jeder Zeile, die eine bestimmte Zeichenfolge (außerhalb eines Kommentars bzw. Strings) enthält – wird ein sich dort befindlicher Kommentar entfernt. Steht an dieser Stelle kein Kommentar, bleibt das Programm unverändert.
- Objekte des Typs *Stripper* entfernen durch *pretty* alle Kommentare aus dem Programm.
- Objekte des Typs *Purifier* sorgen dafür, dass alle Kommentare in eigenen Zeilen stehen und ein bestimmtes, gleichartiges Aussehen haben. Einzeilige Kommentare sehen so aus:

  ```
  /* Textzeile */
  ```

  Mehrzeilige Kommentare haben diese Form

  ```
  /* Textzeile 1
   * Textzeile 2, ...
   */
  ```

  In beiden Fällen ist das erste Zeichen “/” genau so weit eingerückt wie das erste Zeichen der vorangehenden Zeile (abgesehen von Leerzeilen bzw.

White-Space).
- Objekte vom Typ *AltPurifier* ähneln *Purifier*, jedoch gibt es (neben der Darstellung wie in *Purifier*) zwei weitere alternative Darstellungsformen von Kommentaren. Eine Darstellungsform verwendet ausschließlich Zeilen-Kommentare der Form

  ```
  // Textzeile
  ```

  und die andere Darstellungsform Zeilen-Kommentare für einzeilige Kommentare und solche wie in *Purifier* für mehrzeilige Kommentare. In allen Formen stehen Kommentare in eigenen Zeilen und sind gleich tief eingerückt wie die vorige nichtleere Zeile. Die Auswahl der Darstellungsform wird ausschließlich über einen Parameter des Konstruktors getroffen.
- Objekte vom Typ *DeepPurifier* ähneln denen von *AltPurifier*, jedoch ist die Tiefe der Einrückung des Kommentars gegenüber der vorigen nichtleeren Zeile von einem Parameter abhängig. Der Wert dieses Parameters kann jederzeit geändert werden und wirkt sich auf den nächsten Aufruf von *pretty* aus.
- Objekte vom Typ *Composer* vereinfachen die aufeinanderfolgende Anwendung mehrerer Objekte des Typs *Prettifier*. Es gibt eine interne Liste von *Prettifier*-Objekten, die bei einem Aufruf von *pretty* der Reihe nach abgearbeitet werden. Bei leerer Liste gibt ein Aufruf von *pretty* ein unverändertes Programm zurück. Methoden zum Hinzufügen und Entfernen von *Prettifier*-Objekten an/von verschiedenen Positionen innerhalb der Liste erlauben die Anpassung, die sich auf den nächsten Aufruf von *pretty* auswirkt.

Versehen Sie (abstrakte) Klassen und Interfaces mit allen notwendigen Zusicherungen und stellen Sie sicher, dass Sie nur dort eine Vererbungsbeziehung (*extends* oder *implements*) verwenden, wo tatsächlich eine Untertypbeziehung auch hinsichtlich der Zusicherungen besteht. Ermöglichen Sie Untertypbeziehungen zwischen allen diesen Typen, außer wenn Untertypbeziehungen den Beschreibungen der Typen widersprechen würden. Falls zwischen zwei Typen keine Untertypbeziehung besteht, geben Sie in einem Kommentar in der Datei *Test.java* eine Begründung dafür an. Bitte geben Sie eine textuelle Begründung; auskommentierte Programmzeilen reichen dafür nicht.

Sie können so viele zusätzliche (abstrakte) Klassen und Interfaces einführen, wie Sie als vorteilhaft erachten. Die Typstruktur soll trotzdem möglichst einfach und klein bleiben, wobei jedoch alle oben genannten Typen (mit den vorgegebenen Namen) vorkommen müssen, auch solche, die Sie vielleicht für nicht nötig erachten.

Schreiben Sie eine Klasse *Test* zum Testen Ihrer Lösung. Erzeugen Sie Instanzen der oben genannter Typen. Überprüfen Sie so gut Sie können mittels Testfällen, ob dort, wo Sie eine Untertypbeziehung annehmen, Ersetzbarkeit

gegeben ist.

Daneben soll die Klasse *Test.java* als Kommentar eine kurze, aber verständliche Beschreibung der Aufteilung der Arbeiten auf die einzelnen Gruppenmitglieder enthalten – wer hat was gemacht.

Wie die Aufgabe zu lösen ist:

Die insgesamt 100 für diese Aufgabe erreichbaren Punkte sind folgendermaßen auf die zu erreichenden Ziele aufgeteilt:

Untertypbeziehungen richtig erkannt und eingesetzt, fehlende Untertypbeziehungen richtig beschrieben:	50 Punkte
Zusicherungen richtig und sinnvoll eingesetzt:	25 Punkte
Lösung sinnvoll getestet:	10 Punkte
Geforderte Funktionalität vorhanden (so wie in Aufgabenstellung beschrieben):	15 Punkte

Fehler in Untertypbeziehungen sowie ungeeignete Zusicherungen führen daher zu einem sehr hohen Punkteverlust. Obwohl für das Testen der Lösung nur 10 Punkte veranschlagt sind, kann unzureichendes Testen doch zu größerem Punkteverlust führen, wenn dadurch bestehende Mängel nicht rechtzeitig erkannt werden. Auch Mängel in der Funktionalität können einen Verlust von deutlich mehr als 15 Punkten bedeuten, weil diese sehr wahrscheinlich auch aus Fehlern in Untertypbeziehungen und Zusicherungen resultieren.

Die größte Schwierigkeit liegt darin, *alle* Untertypbeziehungen zu finden und Ersetzbarkeit sicherzustellen. Vererbungsbeziehungen, die nicht gleichzeitig auch Untertypbeziehungen sind, führen zu sehr hohem Punkteverlust. Ebenso gibt es hohe Punkteabzüge für nicht wahrgenommene Gelegenheiten, Untertypbeziehungen zwischen den Untertypen von *Prettifier* herzustellen, sowie für fehlende oder falsche Begründungen für nicht bestehende Untertypbeziehungen. Geeignete Begründungen wären etwa Gegenbeispiele, welche Verletzungen der Ersetzbarkeit aufzeigen.

Eine Grundlage für das Auffinden der Untertypbeziehungen sind gute Zusicherungen. Wesentliche (aber nicht alle) Zusicherungen kommen bereits in obigen Beschreibungen der benötigten Typen vor. Sie brauchen diese Beschreibungsteile nur mehr richtig zuzuordnen. Untertypbeziehungen ergeben sich aus den erlaubten Beziehungen zwischen Zusicherungen in Unter- und Obertypen. Es hat sich als günstig erwiesen, alle Zusicherungen, die in einem Obertyp gelten, im Untertyp direkt bei den betroffenen Methoden nochmals hinzuschreiben, da sie sonst leicht übersehen werden.

Vergewissern Sie sich der Korrektheit der Untertypbeziehungen zusätzlich über geeignete Testfälle. Die Anzahl der Testfälle ist nicht entscheidend, wohl aber

deren Qualität: Es kommt darauf an, dass die Testfälle mögliche Verletzungen der Ersetzbarkeit aufdecken können. Umgekehrt sollen Sie sich auch vergewissern, dass Sie keine Gelegenheit für Untertypbeziehungen verpasst haben, indem Sie Beispiele dafür finden, wie angenommene Untertypbeziehungen das Ersetzbarkeitsprinzip verletzen würden.

Zusicherungen in Testklassen werden aus praktischen Überlegungen bei der Beurteilung nicht berücksichtigt. Sorgen Sie aber bitte dafür, dass ein Aufruf von *java Test* einigermaßen nachvollziehbaren Output generiert.

Zur Lösung dieser Aufgabe müssen Sie Untertypbeziehungen und vor allem den Einfluss von Zusicherungen auf Untertypbeziehungen im Detail verstehen. Holen Sie sich entsprechende Informationen aus Kapitel 2 des Skriptums. Folgende zusätzlichen Informationen könnten hilfreich sein:

- Konstruktoren werden in einer konkreten Klasse aufgerufen und sind daher vom Ersetzbarkeitsprinzip nicht betroffen. Konstruktoren haben wie statische Methoden keinen direkten Einfluss auf Untertypbeziehungen.
- Zur Lösung der Aufgabe benötigen Sie keine Exceptions. Sollten Sie dennoch Exceptions verwenden, bedenken Sie, dass eine Instanz eines Untertyps nur dann eine Exception werfen darf, wenn man auch von einer Instanz eines Obertyps in derselben Situation diese Exception erwarten würde.
- Mehrfachvererbung gibt es nur auf Interfaces. Sollte einer der verlangten Typen mehrere Obertypen haben, müssen alle Obertypen bis auf einen Interfaces sein. Die Obertypen sollen auch in diesem Fall so heißen wie in der Aufgabenstellung. Für die Klassen, welche diese Interfaces implementieren, sind dann andere Namen zu wählen.

Schalten Sie Warnungen des Compilers in keinem Fall aus. Sehr viele Warnungen haben mit möglichen Verletzungen der Ersetzbarkeit zu tun.

Lassen Sie sich von der Form der Beschreibung der benötigten Typen nicht täuschen. Daraus, dass die Beschreibung eines Typs die Beschreibung eines anderen Typs teilweise wiederholt, folgt noch keine Ersetzbarkeit. Generell sind Sie wahrscheinlich auf dem falschen Weg, wenn es den Anschein hat, A könne Untertyp von B und B gleichzeitig Untertyp von A sein – außer wenn A und B gleich sind.

Achten Sie auf richtige Sichtbarkeit. Alle oben beschriebenen Typen und Methoden sollen überall verwendbar sein. Die Sichtbarkeit von Implementierungsdetails und insbesondere von Variablen soll aber so stark wie möglich eingeschränkt werden. Bedenken Sie, dass sichtbare Implementierungsdetails die Ersetzbarkeit beeinflussen können.

Was man generell beachten sollte:

Es werden keinerlei Ausnahmen bezüglich des Abgabetermins gemacht. Was nicht rechtzeitig im Repository am Übungsrechner steht, wird bei der Beurteilung nicht berücksichtigt. Da es ab jetzt um 100 Punkte pro Aufgabe geht, wäre jede Ausnahme anderen Gruppen gegenüber ungerecht.

Schreiben Sie nicht mehr als eine Klasse in jede Datei (ausgenommen geschachtelte Klassen), halten Sie sich an übliche Namenskonventionen in Java (Großschreibung für Namen von Klassen und Interfaces, kleine Anfangsbuchstaben für Variablen und Methoden, etc.), und verwenden Sie die Namen, die in der Aufgabenstellung vorgegeben sind. Damit erhöhen Sie die Lesbarkeit Ihrer Programme ganz wesentlich, wodurch bestimmte Fehler in der Beurteilung deutlich unwahrscheinlicher werden.

Übernehmen Sie das vorgegebene Interface *Prettifier* bitte unverändert, einschließlich der Kommentare.

Warum die Aufgabe diese Form hat:

Im Gegensatz zu vielen anderen Aufgaben ist diese Aufgabe ziemlich klar spezifiziert. Der Grund liegt darin, dass die Beschreibungen der Typen nur wenig Interpretationsspielraum bezüglich der Ersetzbarkeit bieten sollen. Die Aufgabe ist so formuliert, dass Untertypbeziehungen (abgesehen von Typen, die Sie vielleicht zusätzlich einführen) eindeutig sind. Über Testfälle und Gegenbeispiele sollten Sie in der Lage sein, große Fehler in der Struktur der Lösung selbst zu finden. Eine Voraussetzung für das Erkennen der richtigen Lösung und deren Eindeutigkeit ist aber ein gutes Verständnis der Ersetzbarkeit. Wenn Ihnen mehrere Lösungsmöglichkeiten (hinsichtlich der Struktur ohne eigene Typen) als gleichermaßen richtig erscheinen, sollten Sie noch einmal ins Skriptum schauen.