UNIVERSITÄT DUISBURG ESSEN

NES Networked Embedded Systems

Miniprojekt 2: Klassen, Schnittstellen und Rekursion

Aufgabe 0 – Datentypen (DIY, d.h. ohne Jack-Tests)

Im Folgenden geben wir Ihnen ein paar „Einzeiler“, die der Variable „result“ einen Wert zuweisen. Notieren Sie sich für jede Zeile den Wert der Variable „result“ und wählen Sie einen möglichst kleinen Datentyp, der dieses Ergebnis **ohne explizite Typumwandlung** speichern kann. Prüfen Sie danach Ihre Berechnungen und Annahmen z.B. mit Hilfe einer main-Methode, in die Sie diese Einzeiler (nach und nach mit entsprechender Deklaration von „result“) einfügen und anschließend den Wert von „result“ ausgeben. Falls das Ergebnis Ihrer Überlegungen nicht mit den Werten übereinstimmt, überlegen Sie sich, wo Ihr Fehler lag.

Tipp: Beachten Sie vor allem die Präzedenz von Operatoren sowie die Tatsache, dass das Konvertieren von größeren Datentypen in kleinere (zum Schutz vor schwerwiegenden Programmierfehlern) eine explizite Typumwandlung erfordert.

Vereinfachung: Die Datentypen „short“ und „byte“ können Sie für die Bearbeitung dieser Aufgabe auf „int“ abbilden. D.h. sie sollen sich primär überlegen ob es sich bei den Zahlen um int, long, float oder double handelt und nicht ob die ganzzahligen Ergebnisse noch in ein Byte bzw. zwei Bytes passen oder ob sie schon 4 Bytes zur Darstellung brauchen.

Beispiel:

result = 1 + 2; (kleinster Datentyp für result ist int (ohne Vereinfachung wäre es byte) , Wert ist 3)

Aufgaben:

result = 1 + 5 * 3;

result = (1 + 5) * 3;

result = 1 + 2.0;

result = 1 + 2.0f;

result = 2 / 4;

result = 3 % 2;

result = -(1 * -1);

result = 1 + 2 /(double)4;

result = 1 + 2 / 4d;

result = Math.sqrt(9) / 2; (Math.sqrt approximiert die Quadratwurzel für eine beliebige Eingabe >= 0)

result = !(2 < 3);

result = 4000l; (viertausend und ein kleines L)

result = 4000l * 10; (viertausend und ein kleines L mal zehn)

result = “Hello World” + “!”;

result = new int[] { 1, 2, 3 };

result = new Object();

Für Fortgeschrittene:

result = 1 + 2147483647;

result = 1 << 1;

result = 3 & 1;

result = 3 ^ 1;

result = ((int)(1.0 / 2.0)) + 3;

result = (int)1.0 / 2.0 + 3.0;

Aufgabe 1 – Klassen und Schnittstellen

In dieser Aufgabe geht es um die Nutzung von Klassen und Schnittstellen zur Strukturierung von Programmen. Diese Aufgabe dient auch als Grundlage für Aufgabe 2. D.h. Sie können Aufgabe 2 nicht ohne die Bearbeitung von Aufgabe 1 beginnen.

Anwendungsbereich des Miniprojekts ist das Brettspiel Schach. Keine Sorge, Sie müssen das Spiel nicht kennen. Für den Moment müssen Sie nur wissen, dass das Spiel auf einem quadratischen Spielfeld – dem Schachbrett – abläuft, das aus schwarzen und weißen Feldern besteht. Diese sind in einem Karomuster angeordnet (erst weiß, dann schwarz, dann weiß, usw.).

In Ihrem Projekt finden Sie 7 Klassen und eine Schnittstelle (Visitor). Die Klassen enthalten bereits Blockkommentare im Javadoc Format und Zeilenkommentare mit „TODO“ Tags. Diese sind als Hilfestellung gedacht. Wenn Sie die Kommentare nicht hilfreich finden, löschen oder ignorieren Sie sie einfach. Der Java Compiler macht das auch.

Die Schnittstelle müssen Sie nicht ändern. Die Klassen sollen Sie wie folgt anpassen:

Field

Diese Klasse repräsentiert ein einzelnes Feld auf dem Schachbrett. Da wir sowohl schwarze als auch weiße Felder haben, definiert Sie eine abstrakte getColor-Methode, die konkrete Felder (für Schach also weiße oder schwarze) implementieren sollen. Hauchen Sie der Klasse mehr Leben ein indem Sie die folgenden Erweiterungen implementieren:

1. Zwei private member Variablen vom Typ int mit den Namen x und y, die die Position des Felds auf dem Brett angeben.
2. Eine private member Variable vom Typ boolean mit dem Namen marked, mit der Sie später Felder markieren können.
3. Einen öffentlichen Konstruktor, der je einen Wert für x und y (in dieser Reihenfolge) annimmt und die entsprechenden privaten Felder setzt.
4. Einen öffentlichen Setter (setMarked) für die marked Variable, mit dem Sie den Wert von marked verändern können.
5. Einen öffentlichen Getter (isMarked) für die marked Variable, der den Wert von marked zurückgibt.
6. Eine Methode, die toString in java.lang.Object überschreibt und den Zustand eines Field-Objekts im folgenden Format zurückliefert: (x,y,marked,color). Also Klammer auf, dann den Wert von x, dann ein Komma, dann den Wert von y, dann ein Komma, usw.

WhiteField und BlackField

Diese Klassen repräsentieren weiße und schwarze Felder. Damit wir nicht alles mehrfach implementieren müssen, wollen wir die Implementierung der Klasse Field nutzen, um uns Tipparbeit zu sparen und die Wartbarkeit des Programms zu verbessern. Dazu machen Sie folgendes:

1. Verändern Sie die Vererbungsbeziehung der Klassen, so dass diese nicht von java.lang.Object sondern von der Klasse Field erben.
2. Der Compiler wird sie jetzt darüber informieren, dass es zwei Probleme gibt:
    a. Die Superklasse hat keinen parameterlosen Default-Konstruktor (mehr), weil Sie zuvor nur einen mit zwei in Parametern in der Klasse Field definiert haben. Deshalb kann der Java Compiler nicht mehr automatisch einen parameterlosen Default-Konstruktor in WhiteField und BlackField erzeugen (denn er weiß ja nicht, was ein sinnvoller Wert für x und y sein würde).

   b. Die Superklasse definiert eine abstrakte getColor Methode, die in allen konkreten (d.h. nicht-abstrakten) Ableitungen oder einer ihrer Superklassen enthalten sein muss. Da WhiteField und BlackField direkt von der abstrakten Klasse Field erben, müssen Sie also entweder selbst abstrakt sein oder die abstrakte Methode konkretisieren (d.h. eine Implementierung liefern).
3. Beheben Sie beide Probleme wie folgt:
   a. Implementieren Sie in jeder Klasse einen Konstruktor, der ebenfalls zwei int Werte (für x und y – in dieser Reihenfolge) annimmt und an die Superklasse zur weiteren Behandlung weiterreicht.
   b. Implementieren Sie die getColor Methode in jeder Klasse, so dass ein WhiteField immer den String „w“ zurückgibt und ein BlackField immer „b“ zurückgibt.

Chessboard

Die Klasse Chessboard repräsentiert das Schachbrett. Schauen Sie sich die Klasse einmal an. Wie Sie sehen, gibt es bereits ein 2-dimensionales Array mit dem Namen fields, das die Felder des Schachbretts enthalten soll. Außerdem gibt es in der Klasse bereits einen Konstruktor, der ein (rechteckiges) Schachbrett beliebiger Breite und Höhe mit einem Karomuster aus schwarzen und weißen Feldern erzeugt. Damit ist der Konstruktor deutlich generischer als das Spiel Schach, denn ein Schachbrett ist immer 8 x 8 Felder groß.

1. Um die bereits vorhandene Flexibilisierung nicht rückgängig zu machen, fügen Sie bitte einen parameterlosen Konstruktor hinzu, der ein Schachbrett der üblichen Größe (d.h. width=8, height=8) erzeugt. Sie können entweder den vorhandenen Code kopieren (ganz schlecht für die Wartbarkeit, bitte machen Sie das nicht und vergessen Sie diesen Vorschlag unverzüglich) oder Constructor Chaining verwenden und den generischen Konstruktor mit den richtigen Werten aufrufen (machen Sie es so).
2. Implementieren Sie jetzt noch die folgenden Hilfsfunktionen für die nächste Aufgabe:
   a. Eine öffentliche Methode hasField(x, y), die wahr genau dann zurückliefert, falls die Position (x, y) ein Feld auf dem Schachbrett ist, ansonsten falsch.
   b. Eine öffentliche Methode getField(x, y), die das mit der Position (x,y) adressierte Feld des Schachbretts zurückliefert, falls (x, y) eine gültige Position ist und die „null“ zurückliefert, falls (x,y) außerhalb des Schachbretts liegt. Falls Sie aufgepasst haben, werden Sie bemerken, dass die „falls-ansonsten-Aussage“ der „hasField(x,y)“ Aussage entspricht. Benutzen Sie doch einfach hasField(x,y) um die Entscheidung in getField(x,y) zu treffen. Dann haben Sie nur noch eine Fehlerquelle und müssen mögliche „ArrayIndexOutOfBounds“-Probleme nur an einer Stelle suchen.

Nachdem das geschafft ist, wollen wir noch verschiedene Ausgabefunktionen für das Schachbrett implementieren. Diese sollen verschiedene Teile des Zustands aller Felder ausgeben. Eine einfache Möglichkeit hierfür wäre das Schreiben von verschachtelten for-Schleifen in jeder Ausgabemethode, die in geeigneter Weise über alle Felder in fields iteriert. Da es in dieser Aufgabe aber nicht um Arrays sondern um Klassen und Schnittstellen geht, machen wir das aber heute mal anders.

Intermezzo: Visitor

Schauen Sie sich dazu zunächst einmal die Schnittstelle mit dem Namen „Visitor“ an. Diese definiert zwei Methoden:

- Die Methode nextField erhält als Parameter ein Objekt vom Typ Field und liefert keinen Rückgabewert. Sie soll uns in wenigen Minuten bei der Ausgabe einen bestimmten Teilzustand des übergebenen „field“ dienen.

- Die Methode nextRow weist keine Parameter auf und hat keinen Rückgabewert. Diese Methode werden wir nutzen, um an der richtigen Stelle einen Zeilenumbruch zu erzeugen.

Zurück zum: Chessboard

Wenden Sie jetzt Ihre Aufmerksamkeit der Methode „visitFields“ in der Klasse Chessboard zu und überlegen Sie sich, was diese Methode macht. Das bedeutet Sie sollten jetzt aufhören hier zu lesen und sich die Klasse anschauen. Es nützt Ihnen mehr, wenn Sie versuchen den Code zu verstehen, bevor Ihnen dieses Übungsblatt die Erklärung gibt. So jetzt bitte zur Klasse wechseln und die Programmlogik verstehen.

Zu Beginn gibt es eine Anweisung, die prüft ob das Spielfeld überhaupt ein Feld hat. Das ist erforderlich da der Aufruf von „fields[0].length“ sonst ein nicht vorhandenes Array (am Index 0, den es nicht gibt) referenzieren würde. Falls nicht, wird die Methode mittels „return“ beendet bevor fields[0].length erreicht wird.

Ansonsten findet eine Iteration (mittels 2 verschachtelter for-Schleifen) statt, die die Felder in „fields“ Zeile für Zeile abarbeitet. In der inneren Schleife wird auf dem als Methodenparameter übergebenen „visitor“ für jedes Feld die „nextField“ Methode aufgerufen. Sobald die innere Schleife abgearbeitet wurde (und x am Zeilenende angekommen ist), wird vor der Bearbeitung der nächsten Zeile durch die äußere „nextRow“ aufgerufen. D.h. für ein 2x2 Schachbrett würde der Visitor folgende Aufrufe erhalten:

1. nextField(fields[0][0]); // erste Zeile, erste Spalte
2. nextField(fields[0][1]); // erste Zeile, zweite Spalte
3. nextRow();
4. nextField(fields[1][0]); // zweite Zeile, erste Spalte
5. nextField(fields[1][1]); // zweite Zeile, zweite Spalte
6. nextRow();

Das können wir jetzt nutzen um mehrere strukturierte Ausgabe zu erhalten, ohne dabei noch Schleifen verwenden zu müssen. Dazu müssen wir lediglich verschiedene Implementierungen der Schnittstelle Visitor entsprechend programmieren.

ColorPrinter und MarkPrinter

Zur Erzeugung der Ausgaben implementieren Sie jetzt die Klassen ColorPrinter und MarkPrinter. Die Klasse ColorPrinter soll die Farben der Felder ausgeben und die Klasse MarkPrinter soll den Zustand der Variable „marked“ ausgeben. Dazu führen Sie folgenden Änderungen durch:

1. Ändern Sie die Typdeklaration der Klassen so, dass diese die Schnittstelle Visitor implementieren.
2. Implementieren Sie die Methode nextRow so, dass diese einen Zeilenumbruch auf der Konsole erzeugt (siehe System.out.println).
3. Implementieren Sie die Methode nextField wie folgt:
    a. ColorPrinter: hier soll die Farbe des übergebenen Felds (ohne Zeilenumbruch, siehe System.out.print) auf der Konsole ausgegeben werden.
    b. MarkPrinter: hier soll der Zustand der marked Variable des Felds (ohne Zeilenumbruch) ausgegeben werden. Um eine möglichst kompakte Darstellung zu erzeugen soll „X“ ausgebgeben falls marked „true“ ist und „-" falls marked „false“ ist.

Zurück zum: Chessboard

Damit wir den Zustand des Chessboard mit Hilfe der Printer ausgeben können, implementieren Sie jetzt noch zwei weitere Methoden im Chessboard:

1. void printMarks(): nutzen Sie hier die visitFields Methode mit einem MarkPrinter um die Ausgabe der Markierungen der Felder des Schachbretts zu erzeugen.
2. void printColors(): nutzen Sie hier die visitFields Methode mit einem ColorPrinter um die Ausgabe der Farben der Felder des Schachbretts zu erzeugen.

Implementieren Sie die statische main-Methode in Chessboard um Ihre bisherige Implementierung zu testen. Erzeugen Sie dazu ein Chessboard und rufen Sie printColors und printMarks auf. Die Ausgabe sollte ungefähr so aussehen:

```
printColors:
wbwbwbwb
bwbwbwbw
wbwbwbwb
bwbwbwbw
wbwbwbwb
bwbwbwbw
wbwbwbwb
bwbwbwbw

printMarks:
--------
--------
--------
--------
--------
--------
--------
--------
```

FieldStringifyer

Weil das so gut gekappt hat, wollen wir jetzt noch die Methode toString() im Chessboard implementieren. Die Methode soll angelehnt an printColors und printMarks einen String erzeugen, der aus den einzelnen Rückgabewerten der toString()-Methoden der Felder zusammengesetzt ist. Im Gegensatz zu ColorPrinter und MarkPrinter soll der String jedoch nicht auf die Konsole geschrieben werden, sondern in einer Variable von FieldStringifyer vorgehalten werden. Machen Sie hierfür die folgenden Änderungen an der Klasse:

- Ändern Sie die Typdeklaration so, dass FieldStringifyer die Schnittstelle Visitor unterstützt.
- Fügen Sie der Klasse eine private Member-Variable mit dem Namen „text“ und dem Typ String hinzu. Die Variable sollte mit einem leeren String initialisiert werden.
- Implementieren Sie einen öffentlichen Getter für „text“.
- Implementieren Sie die Methode nextField so, dass die String-Repräsentation des Felds (also der Rückgabewert von field.toString()) an den Inhalt von „text“ angehängt wird.
- Implementieren Sie die Methode nextRow so, dass diese einen Zeilenumbruch („\n“) an den Inhalt von „text“ anhängt.

Zurück zum: Chessboard

Implementieren Sie jetzt mit Hilfe des FieldStringifyers und der visitFields Methode von Chessboard die „toString()“ Methode, so dass diese den Inhalt von der „text“ Variable des FieldStringifyers nach Traversierung des Chessboards zurückggibt. Testen Sie danach Ihre Implementierung durch einen geeigneten Aufruf von toString() in der main-Methode.

Wenn Sie damit fertig sind, implementieren Sie als Vorbereitung für die nächste Aufgabe noch die Methode „clearMarks“. Diese Methode soll die Markierung aller Felder des Schachbretts (durch

Aufruf der setMarked-Methode mit dem Wert „false“) zurücksetzen. Hierfür müssen Sie über alle Felder des Schachbretts mit Hilfe von Schleifen iterieren.

Für Überflieger: Um die Schleifenlogik der visitFields Methode wiederzuverwenden, können Sie zur Implementierung von „clearMarks“ auch eine anonyme Klasse[1] verwenden, die die Visitor-Schnittstelle implementiert und in nextField das jeweilige Field zurücksetzt. Das Konzept der anonymen Klassen wurde in der Vorlesung bislang jedoch noch nicht behandelt. Falls Sie es sich selbst anschauen wollen und das Konzept verstehen, erkennen Sie ganz sicher die schlichte Eleganz des Visitor-Entwurfsmusters. Falls nicht, ist das auch kein Problem. Entwurfsmuster sind nicht Bestandteil dieser Vorlesung und falls Ihr Studiengang das Wort „Informatik“ enthält, werden Sie sie sicherlich noch in anderen Veranstaltungen kennen lernen.

[1] https://docs.oracle.com/javase/tutorial/java/javaOO/anonymousclasses.html

Aufgabe 2 – Rekursion

Der Springer (engl. Knight) ist eine Schachfigur, die in jedem Zug je nach Position auf dem Schachbrett bis zu 8 verschiedene Bewegungen ausführen kann. Auf Basis der aktuellen Position x, y lassen diese sich wie folgt berechnen:

1. (X+/-2, Y+/-1) → (X+2, Y+1), (X+2, Y-1), (X-2, Y+1), (X-2, Y-1)
2. (X+/-1, Y+/-2) → (X+1, Y+2), (X+1, Y-2), (X-1, Y+2), (X.1, Y-2)

Implementieren Sie mit Hilfe von Rekursion die öffentliche Methode „void markKnightMoves(int x, int y, int n)“ in der Klasse Chessboard, die ausgehend von der Position x, y alle Felder auf dem Schachbrett markiert, die ein Springer in „n“ Zügen erreichen kann.

- Für n<0 soll kein Feld markiert werden.
- Für n=0 soll lediglich die Startposition markiert werden.
- Für n=1 soll die Startposition markiert werden und alle Felder (X+/-2, Y+/-1) und (X+/-1, Y+/-2), die sich auf dem Spielfeld befinden.
- Für n=2 sollen die Felder von n=1 und alle von diesen Feldern mit (X+/-2, Y+/-1) und (X+/-1, Y+/-2) erreichbaren markiert werden, usw.

Beachten Sie bei Ihrer Implementierung, dass der Springer das Spielfeld nicht verlassen darf (hasField(x,y) muss also „wahr“ sein). Testen Sie Ihre Implementierung über eine geeignete Implementierung der main-Methode, die z.B. die Markierungen für verschiedene Startpositionen und Werte für n mit Hilfe von printMarks ausgibt.

Anwendung

Berechnen Sie mit Hilfe Ihres Programms die minimale Anzahl an Zügen, die ein Springer braucht um von der oberen linken Ecke zur unteren rechten Ecke eines Schachbretts (mit normaler Größe) zu gelangen.

Komplexität (zum Nachdenken)

Überlegen Sie sich wie häufig sich die Methode markKnightMoves in Abhängigkeit von n im schlimmsten Fall (d.h. höchstens) selbst erneut aufruft. Wenn Sie sich nicht sicher sind, können Sie Ihr Ergebnis mit Hilfe Ihres Programms selbst testen. Dafür können Sie einfach die Aufrufe von markKnightMoves mit Hilfe einer int-Variable in Chessboard zählen (d.h. für jeden Aufruf einfach 1 addieren und den Wert in der main-Methode ausgeben und ggf. auf 0 zurücksetzen). Um den „schlimmsten Fall“ zu bestimmen, nehmen Sie einfach ein sehr großes Schachbrett (z.B. 100x100 Felder) und starten Sie in der Mitte. Dann sollten für kleine Werte von n alle 8 Züge immer möglich sein (d.h. sie sollten noch auf dem Spielbrett landen).

Zum Üben von Klassen und Schnittstellen (ohne Jack-Tests)

In einem Schachspiel gibt es neben dem Springer noch weitere Schachfiguren. Jede Figur hat jedoch unterschiedliche Spielzüge. Ein Läufer (engl. Bishop) kann nur diagonal ziehen (x+z, y+z), ein Turm (engl. Rook) kann nur entlang der x- oder y-Achse ziehen also nur (x+z, y) oder (x, y+z), usw.

Wir könnten jetzt natürlich für jede dieser Figuren eine „markSomeFigureMoves“ Methode implementieren um noch mehr Funktionen auf unserem Schachbrett zu unterstützen. Alternativ können wir auch das Verhalten der Figur (also in unserem Fall: die möglichen Züge ausgehend von der aktuellen Position) mit Hilfe einer Schnittstelle abstrahieren. Diese Schnittstelle können wir dann mit verschiedenen Klassen (also eine für jede Figur) je nach Zugmöglichkeiten implementieren. Das erlaubt uns dann schließlich die Implementierung einer „markMoves“-Logik unabhängig von den Bewegungsmöglichkeiten einer Figur.

Eine mögliche Vorgehensweise wäre z.B.:

1. Erstellen Sie eine neue Schnittstelle „Figure“ mit der Methode getMoves(int x, int y, int boardWidth, int boardHeight), die ein neues Chessboard der Größe boardWidth x boardHeight erstellt und zurückgibt. Auf diesem Chessboard sollen alle Felder markiert sein, die ausgehend von der Position (x, y) mit der Figur in einem Schritt erreichbar sind (also ähnlich wie markKnightMoves für n=1).
2. Erstellen Sie drei Klassen „Knight“, „Bishop“ und „Rook“ so dass diese die Schnittstelle Figure implementieren und ein entsprechendes Chessboard mit markierten Feldern zurückliefern.
3. Fügen Sie der Klasse Chessboard eine neue Methode markMoves(int x, int y, int n, Figure figure) hinzu, so dass diese mit Hilfe einer beliebigen Figure alle in n Zügen erreichbaren Felder markiert. Hinweise:
    a. In markMoves müssen Sie also getMoves der Figure-Klasse mit geeigneten Parametern aufrufen. Dann erhalten Sie ein Chessboard mit markierten Feldern für den nächsten Zug.
    b. Nun müssen Sie in den „markierten“ Feldern des zurückgegebenen Schachbretts weiter rekursiv absteigen (denn das Schachbrett aus getMoves enthält ja nur einen Zug). Dazu müssen Sie über das Schachbrett iterieren und für jedes markierte Feld die entsprechenden Aktionen (also erst Feld markieren und ggfs. weiter rekursiv absteigen) auslösen.

Testen Sie Ihre Implementierung mit Hilfe einer geeigneten main-Methode.