Separation of Concerns

Gruppengrösse	Einzelarbeit

Die folgenden Aufgaben beziehen sich auf das Repository *SeparationOfConcerns*, welches auf GitHub zu finden ist.

Das Repository besteht aus drei Projekten:

1. **SeparationOfConcerns**: die Programmbibliothek, welche die eigentliche Logik enthält. (Die Programmlogik befindet sich jeweils in einer einzigen Methode, was der Idee von Separation of Concerns widerspricht.)
2. **SeparationOfConcerns.Demo**: das ausführbare Programm, welches die drei Beispiele aufruft. (Kommentieren Sie die Aufrufe in der Main-Methode aus, um sich auf ein bestimmtes Problem konzentrieren zu können.)
3. **SeparationOfConcerns.Test**: das xUnit-Test-Projekt, welches keine Testfälle enthält. (Diese waren offensichtlich zu umständlich zu schreiben.)

1. Anweisungen

Sie müssen folgendes tun:

1. Erstellen Sie einen Fork von diesem Repository. Klonen Sie anschliessend Ihr persönliches Repository.
2. Suchen Sie sich einer der nachfolgend beschriebenen Aufgaben aus. Sie können in beliebiger Reihe vorgehen.
3. Bei jeder Aufgabe gehen Sie in den folgenden Schritten vor:
    1. Modifizieren Sie SeparationsOfConcerns.Demo/Program.cs sodass nur noch der Code zur jeweiligen Aufgabe ausgeführt wird (andere Aufrufe können Sie auskommentieren.)
    2. Testen Sie den jeweiligen Beispielcode mit verschiedenen Argumenten. (Befolgen Sie hierzu die Anweisungen in README.md des Repositories.)
    3. Betrachten Sie den Code zu Ihrer Aufgabe. Überlegen Sie sich, welche verschiedenen Aspekte in der einen grossen Methode gemeinsam behandelt werden. (Tipp: Oft wer- den “Berechnung” und “Ausgabe” vermischt. Es gibt aber noch weitere Aspekte, die man separieren könnte.)
    4. Nehmen Sie ein Refactoring am Code vor, um eine bessere Separation of Concerns zu erreichen. Passen Sie, wenn nötig, den aufrufenden Code im Demo-Projekt an, um den Beispielcode wieder laufen lassen zu können.
    5. Dank der verbesserten Separation of Concerns sollte sich der Code besser automatisiert testen lassen. Schreiben Sie Unittests für Ihren überarbeiteten Code. (Entscheiden Sie selber welche und wie viele.)
4. Wiederholen Sie dieses Vorgehen für die nächste Aufgabe.

Die einzelnen Aufgaben werden in den folgenden Abschnitten erklärt.

2. Multiplikationstabelle

Die statische Methode *MultiplicationTable*.For gibt eine Multiplikationstabelle für die Zahlen des Listenparameters aus. Es wird eine Matrix ausgegeben, bei der jede Zahl als Spalte und Zeile vorkommt. Für jede Spalten- und Zeilenkombination wird das Produkt berechnet und in der jeweiligen Zelle ausgegeben.

Die Ausgabe mit der korrekten Spaltenbreite ist dabei recht ausgeklügelt. Weniger gut gelöst ist die Organisation des Codes. Verbessern Sie diese.

3. Primfaktorzerlegung

Bei der Primfaktorzerlegung wird eine Zahl in ihre *Primfaktoren* zerteilt. Man beginnt mit der kleinsten Primzahl 2 und versucht die gegebene Zahl dadurch zu teilen. Ist eine Teilung möglich, wird wiederum versucht den Rest durch die gleiche Primzahl zu teilen. Ist die Teilung nicht restlos möglich, wird mit der nächsten Primzahl fortgefahren. Die Faktoren, mit denen die Teilung restlos

funktioniert hat, werden dabei aufgelistet. Multipliziert man diese Faktoren anschliessend, kommt man wieder auf die Originalzahl.

Beispiele für Primfaktorzerlegungen:

Zahl	Primfaktoren	Kontrolle
10	2, 5	2 * 5 = 10
42	2, 3, 7	2 * 3 * 7 = 42
55	5, 11	5 * 11 = 55
99	3, 3, 11	3 * 3 * 11 = 99
1024	2, 2, 2, 2, 2, 2, 2, 2, 2, 2	2 ^ 10 = 1024

Die Primfaktorenzerlegung wird z.B. bei der Kryptographie (genauer: der Kryptoanalyse) verwendet, um beispielsweise RSA-Schlüssel zu knacken. (Hierbei kommen jedoch sehr grosse Zahlen zum Einsatz.)

Für die Primfaktorzerlegung müssen zunächst die Primzahlen bis zu einer gegebenen Zahl gefunden werden. Dies könnte auch wesentlich effizienter implementiert werden. Versuchen Sie eine Primfaktorzerlegung einer grösseren Zahl, sodass es spürbar langsam läuft. Verbessern Sie anschliessend die Performance, bis das Programm wieder schnell genug läuft (Ideen: Caching oder das *Sieb des Eratosthenes*).

Die Methode *PrimeFactors*. Factor erwartet eine Liste von zu faktorisierenden positiven Zahlen. Verbessern Sie deren Organisation.

4. Monty-Hall-Problem

In der Spielshow *Monty Hall* bekommen die Teilnehmer drei Tore zur Auswahl: Hinter einem verbirgt sich der Hauptgewinn von einem Auto, hinter den anderen beiden eine Ziege als Niete/Trostpreis.

Nachdem der Spieler sich für ein Tor entscheidet, lässt Monty Hall jeweils ein anderes Tor öffnen, hinter welchem sich eine Ziege verbirgt. Der Spieler hat nun die Möglichkeit seine Wahl zu ändern oder bei der ursprünglichen Wahl zu bleiben.

Das *Monty-Hall-Problem* lautet folgendermassen: *Verbessern sich durch das Wechseln der Wahl die Gewinnchancen?*

Im gegebenen Code wird versucht, die Frage mittels einer Simulation zu lösen. Hierbei wird eine gegebene Anzahl von Spielen gespielt. Dabei wird ausgerechnet, wie oft der Spieler gewinnen würde wenn er:

1. bei seiner Wahl bleibt
2. seine Wahl ändert

Das Ergebnis ist verblüffend. Weniger beeindruckend (und schwerer testbar) ist jedoch die Implementierung der Methode MontyHall.Play.