

# Separation of Concerns

---

Aufgabe "Separation of Concerns", Modul 450

Die folgenden Aufgaben beziehen sich auf die Beispiele **SeparationOfConcerns** (die Programmbibliothek, welche die eigentliche Logik enthält), **SeparationOfConcerns.Demo** (das ausführbare Programm, welches die drei Beispiele aufruft), **SeparationOfConcerns.Test** (das xUnit-Test-Projekt, welches keine Testfälle enthält).

## 1 Anweisungen

Du musst folgendes tun:

1. Suche dir eine der nachfolgend beschriebenen Aufgaben aus. Du kannst in beliebiger Reihenfolge vorgehen.
2. Bei jeder Aufgabe gehst du in den folgenden Schritten vor:
  1. Modifiziere `SeparationsOfConcerns.Demo/Program.cs`, sodass nur noch der Code zur jeweiligen Aufgabe ausgeführt wird (andere Aufrufe kannst du auskommentieren.)
  2. Teste den jeweiligen Beispielcode mit verschiedenen Argumenten. (Befolge hierzu die Anweisungen in `README.md` des Repositories.)
  3. Betrachte den Code zu deiner Aufgabe. Überlege dir, welche verschiedenen Aspekte in der einen grossen Methode gemeinsam behandelt werden. (Tipp: Oft werden "Berechnung" und "Ausgabe" vermischt. Es gibt aber noch weitere Aspekte, die man separieren könnte.)
  4. Nimm ein Refactoring am Code vor, um eine bessere *Separation of Concerns* zu erreichen. Passe, wenn nötig, den aufrufenden Code im Demo-Projekt an, um den Beispielcode wieder laufen zu lassen.
  5. Dank der verbesserten *Separation of Concerns* sollte sich der Code besser automatisiert testen lassen. Schreibe Unittests für deinen überarbeiteten Code. (Entscheide selbst welche und wie viele.)
3. Wiederhole 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. Verbessere 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. Versuche eine Primfaktorzerlegung einer grösseren Zahl, sodass es spürbar langsam läuft. Verbessere 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. Verbessere 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`.